REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE 18.Dec.03

3. REPORT TYPE AND DATES COVERED DISSERTATION

4. TITLE AND SUBTITLE
"A SOUNDING ROCKERT ATTITUDE DETERMINATION ALGORITHM SUITABLE FOR IMPLEMENTATION USING LOW COST SENSORS"

5. FUNDING NUMBERS

6. AUTHOR(S)
MAJ CHARLTON MARK C

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
UNIVERSITY OF ALASKA FAIRBANKS

8. PERFORMING ORGANIZATION REPORT NUMBER
CI02-1331

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
THE DEPARTMENT OF THE AIR FORCE
AFIT/CIA, BLDG 125
2950 P STREET
WPAFB OH 45433

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Unlimited distribution
In Accordance With AFI 35-205/AFIT Sup 1

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

20040105 039

14. SUBJECT TERMS

15. NUMBER OF PAGES 321

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT

18. SECURITY CLASSIFICATION OF THIS PAGE

19. SECURITY CLASSIFICATION OF ABSTRACT

20. LIMITATION OF ABSTRACT

A SOUNDING ROCKET ATTITUDE DETERMINATION ALGORITHM SUITABLE FOR IMPLEMENTATION USING LOW COST SENSORS

By

Mark Christopher Charlton

RECOMMENDED:

Charles E Mayer

Joseph S. Hawkins
Advisory Committee Chair

Chair, Department of Electrical and Computer Engineering

APPROVED:

Dean, College of Science, Engineering and Mathematics

Dean of the Graduate School

November 18, 2003
Date

A SOUNDING ROCKET ATTITUDE DETERMINATION ALGORITHM SUITABLE FOR IMPLEMENTATION USING LOW COST SENSORS

A

THESIS

Presented to the Faculty
of the University of Alaska Fairbanks

in Partial Fulfillment of the Requirements
for the Degree of

DOCTOR OF PHILOSPHY

By

Mark Christopher Charlton, B.S., M.S.

Fairbanks, Alaska

December 2003

The views expressed in this article are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U. S. Government.

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

Abstract

The development of low-cost sensors has generated a corresponding movement to integrate them into many different applications. One such application is determining the rotational attitude of an object. Since many of these low-cost sensors are less accurate than their more expensive counterparts, their noisy measurements must be filtered to obtain optimum results. This work describes the development, testing, and evaluation of four filtering algorithms for the nonlinear sounding rocket attitude determination problem. Sun sensor, magnetometer, and rate sensor measurements are simulated. A quaternion formulation is used to avoid singularity problems associated with Euler angles and other three-parameter approaches. Prior to filtering, Gauss-Newton error minimization is used to reduce the six reference vector components to four quaternion components that minimize a quadratic error function. Two of the algorithms are based on the traditional extended Kalman filter (EKF) and two are based on the recently developed unscented Kalman filter (UKF). One of each incorporates rate measurements, while the others rely on differencing quaternions. All incorporate a simplified process model for state propagation allowing the algorithms to be applied to rockets with different physical characteristics, or even to other platforms. Simulated data are used to develop and test the algorithms, and each successfully estimates the attitude motion of the rocket, to varying degrees of accuracy. The UKF-based filter that incorporates rate sensor measurements demonstrates a clear performance advantage over both EKFs and the UKF without rate measurements. This is due to its superior mean and covariance propagation characteristics and the fact that differencing generates noisier rates than measuring. For one sample case, the "pointing accuracy" of the rocket spin axis is improved by approximately 39 percent over the EKF that uses rate measurements and by 40 percent over the UKF without rates. The performance of this UKF-based algorithm is evaluated under other-than-nominal conditions and proves robust with respect to data dropouts, motion other than predicted and over a wide range of sensor accuracies. This UKF-based algorithm provides a viable low cost alternative to the expensive attitude determination systems currently employed on sounding rockets.

Table of Contents

Signature Page i

Title Page ii

Abstract iii

Table of Contents iv

List of Figures ix

List of Tables xii

List of Appendices xiii

Acknowledgements xiv

1.0 Introduction 1

1.1 Background 1

1.2 Potential Applications 2

1.3 Research Questions And Contributions of This Work 3

2.0 Previous Work 5

2.1 Solutions for Low-Cost Applications in General 5

2.2 Work on Rockets 6

3.0 Suitable Coordinate Axes 8

3.1 Introduction 8

3.2 Common Coordinate Frames 8

3.3 Frames Best Suited to this Problem: Navigation Frame and Body Frame 9

4.0 Coordinate Parameterizations 13

4.1 Introduction 13

4.2 Common Coordinate Parameterizations 13

4.3 Parameterization Best Suited to This Problem: Quaternions 15

5.0 Rocket Rotational Dynamics 22

5.1 Introduction 22

5.2 Basics of Rocket Rotational Motion 22

5.3 Simplified Model of Rocket Motion 26

6.0 Attitude Sensing 28

6.1 Introduction 28

6.2 Common Attitude Sensors 28

6.3 Sensors Best Suited to This Problem: Sun Sensor, Magnetometer, Rate Gyro 32

7.0 Applicable Sensor Fusion Principles 34

7.1 Introduction 34

7.2 Sensor Fusion Techniques: Weighted Averaging, Error Checking, Error Minimization / Parameter Optimization 34

7.3 Wahba's Problem and Gauss-Newton Iteration 39

8.0 Applicable Filtering Techniques 41

8.1 Introduction 41

8.2 Promising Filtering Techniques 41

8.2.1 Extended Kalman Filter 41

8.2.2 Variations on the Extended Kalman Filter: Adaptive Kalman Filter / Multiple Model Filter 46

8.2.3 Alternatives to the Extended Kalman Filter 47

8.3 Most Suitable Filtering Technique for this Application: Unscented Kalman Filter 51

9.0 Designing the Algorithm 52

9.1 Restating the Objective 52

9.2 Overview of the Approach 52

9.2.1 Measurements Available to the Algorithm 53

9.2.2 Processing Methods 54

9.2.3 Block Diagram of the Overall Process 55

9.3 Sensor Characteristics 56

9.3.1 Sun Sensor 56

9.3.2 Magnetometer 56

9.3.3 Rate Gyroscopes 56

9.4 Dynamic Modeling of Rocket Motion 57

9.4.1 Simplified Dynamic Model 57

9.4.2 Model Parameters 58

9.5 Quaternion-Based Attitude 58

9.5.1 Relationship Between Quaternions and Euler Angles 58

9.5.2 Relationship Between Quaternion Rates and Body Rates 59

9.6 Gauss-Newton Parameter Optimization 60

9.6.1 Design of the Gauss-Newton Routine 61

9.6.2 Inputs to Error Minimization Routine 65

9.7 Extended Kalman Filter Design 65

9.7.1 Choice of State Vector 67

9.7.2 Derivation and Definition of Filter Matrices 67

9.7.2.1 Initializing the State Vector, $\hat{x}_0$ 67

9.7.2.2 Initializing the Covariance Matrix , P 68

9.7.2.3 Computing the Measurement Noise Matrix, R 68
9.7.2.4 Computing the State Transition Matrix, Φ 70
9.7.2.5 Computing the Process Noise Matrix, Q 76
9.7.2.6 Computing the Covariance Matrix Before the Update, M 79
9.7.2.7 Propagating the State Vector Forward, $\bar{x}$ 80
9.7.2.8 Calculating the Projected Observations, $\bar{z}$ 80
9.7.2.9 Calculating the Measurement Matrix, H 80
9.7.2.10 Computing the Kalman Gain Matrix, K 80
9.7.2.11 Computing the Covariance Matrix After the Update, P 81
9.7.2.12 Computing the New State Estimate, $\hat{x}$ 81
9.8 Unscented Kalman Filter Design 81
9.8.1 Choice of State vector 83
9.8.2 Derivation and Definition of Filter Matrices 83
9.8.2.1 Initializing the State Vector, $\hat{x}_0$ 83
9.8.2.2 Initializing the Covariance Matrix , P 83
9.8.2.3 Computing the Measurement Noise Matrix, R 84
9.8.2.4 Computing the Process Noise Matrix, Q 84
9.8.2.5 Calculating the Sigma Points, χ_i 85
9.8.2.6 Propagating the Sigma Points Forward in Time, $\bar{\chi}_i$ 86
9.8.2.7 Computing the Predicted Mean for Each State, $\bar{x}$ 87
9.8.2.8 Computing the Covariance Matrix Before the Update, M 87
9.8.2.9 Augmenting the Sigma Point Matrix, $\bar{\chi}_i$ 87
9.8.2.10 Propagating Prediction Points Forward in Time, $\bar{Z}_i$ 89
9.8.2.11 Computing the Predicted Observations, $\bar{z}$ 89
9.8.2.12 Calculating the Innovation Covariance, P_{zz} 89
9.8.2.13 Calculating the Cross Correlation Matrix, P_{xz} 90
9.8.2.14 Computing the Kalman Gain Matrix, K 90
9.8.2.15 Computing the Covariance Matrix After the Update, P 90
9.8.2.16 Computing the New State Estimate, $\hat{x}$ 90
10.0 Tuning of the Algorithm and Validation of Performance 91
10.1 Computer Simulation of True Motion 91
10.1.1 Methodology for Simulating Motion 91
10.1.2 Single 90 Degree Rotation 93

10.1.3 Two Successive 90 Degree Rotations 96

10.1.4 Single 120 Degree Rotation 97

10.2 Computer Simulation of Noisy Sensor Measurements 100

10.2.1 Corrupting "Truth" with Noise to Simulate Sensor Measurements 100

10.2.2 Simulating a Sensor or Measurement Bias 101

10.2.3 Simulating Data "Dropouts" 101

10.3 Measurement of Algorithm Performance 101

10.3.1 Attitude Estimation Performance 101

10.3.2 Rotational Rate Estimation Performance 102

10.3.3 Overall Figure of Merit 102

10.4 Error Minimization via Gauss-Newton 102

10.4.1 Convergence Tolerance and the Optimum Number of Steps 103

10.4.2 Convergence of Error Minimization Routine 104

10.5 Demonstrated Performance of the Extended Kalman Filter 107

10.5.1 "Tuning" the Extended Kalman Filter 107

10.5.1.1 Choosing an Integration Scheme 108

10.5.1.2 Setting Process Noise, Φ_S 110

10.5.1.3 Setting Motion Time Constants, τ_r 112

10.5.2 Sample Performance When no Rate Measurements are Available 113

10.5.3 Sample Performance When Rate Measurements are Available 114

10.6 Demonstrated Performance of the Unscented Kalman Filter 115

10.6.1 "Tuning the Unscented Kalman Filter 116

10.6.1.1 Choosing an Integration Scheme 116

10.6.1.2 Setting Process Noise, Φ_S 117

10.6.1.3 Setting the Time Constants of Motion, τ_r 118

10.6.1.4 Choosing the UT Parameters 118

10.6.2 Sample Performance When no Rate Measurements are Available 120

10.6.3 Sample Performance When Rate Measurements are Available 122

10.7 Summary of Filter Relative Performance 124

11.0 Filter Behavior Under Other Than Nominal Conditions 128

11.1 Performance With Less Frequent Measurements 128

11.2 Performance With Sensors of Greater and Lesser Accuracy 130

11.3 Behavior in the Event of Loss of Measurements 131

11.4 Impact of Sensor Bias 138

11.5 Performance When Actual Motion is Different Than Predicted Motion 140
11.6 Determining if the Filter is Operating Correctly 142
11.7 A Discussion of Error in the Estimate 151
12.0 Conclusions 153
12.1 Significant Conclusions 153
12.2 Suggestions for Further Research 154
13.0 References 156
13.1 Literature Cited 156
13.2 Other References 161

List of Figures

Figure 3.1: Earth-Centered-Inertial Coordinate Frame (Figure 2.2-2 in [28]) 11
Figure 3.2: Representative Body-Fixed Coordinate Frame (Figure 7.9-1 in [30]) 12
Figure 4.1: Euler Rotations of a Rigid Body (Figure1.8 in [40]) 15
Figure 5.1: White Noise Driven Angular Rates (After Figure 3.1 in [11]) 27
Figure 8.1: Example of the UT for Mean and Covariance Propagation (Figure 7.3 in [68]) 50
Figure 9.1: Overview of Attitude Estimation Algorithm 55
Figure 9.2: White Noise Driven Angular Rates (After Figure 3.1 in [11]) 57
Figure 9.3: Process Model Relating Unit Quaternion to White Noise Generated Angular Rates 60
Figure 9.4: Outline of Gauss-Newton Routine 65
Figure 9.5: Overall Block Diagram of Extended Kalman Filter 66
Figure 9.6: Overall Block Diagram of Unscented Kalman Filter 82
Figure 10.1: Generation of Simulated Data and Measurements 92
Figure 10.2: Single 90 Degree Rotation as Depicted by Attitude_sim.m 94
Figure 10.3: Quaternion Elements for Single 90 Degree Rotation as Produced by Attitude_sim.m 95
Figure 10.4: Euler Angles for Single 90 Degree Rotation as Produced by Attitude_sim.m 95
Figure 10.5: Successive 90 Degree Rotations as Depicted by Attitude_sim.m 96
Figure 10.6: Quaternion Elements for Successive 90 Degree Rotations as Produced by Attitude_sim.m. 97
Figure 10.7: Euler Angles for Successive 90 Degree Rotations as Produced by Attitude_sim.m 98
Figure 10.8: Single 120 Degree Rotation About (1,1,1) 98
Figure 10.9: Elements for Single 120 Degree Rotation About (1,1,1) 99
Figure 10.10: Euler Angles for Single 120 Degree Rotation About (1,1,1) 100
Figure 10.11: Single Time Step Error History for Successful Convergence 105
Figure 10.12: Single Time Step Quaternion Component History for Successful Convergence 105
Figure 10.13: Single Time Step Error History for Unsuccessful Convergence 106
Figure 10.14: Single Time Step Quaternion Component History for Unsuccessful Convergence 107
Figure 10.15: Lag in Filter Rate Estimate When Rate Measurements not Used 122
Figure 10.16: No Lag in Filter Rate Estimate When Rate Measurements are Used 123
Figure 11.1: UKF Estimate of ω_1 Through One Second Measurement Dropout 133
Figure 11.2: Close-up of UKF Estimate of ω_1 Through One Second Measurement Dropout 134
Figure 11.3: UKF Estimate of ω_2 Through One Second Measurement Dropout 134
Figure 11.4: Close-up of UKF Estimate of ω_2 Through One Second Measurement Dropout 135
Figure 11.5: UKF Estimate of ω_3 Through One Second Measurement Dropout 135
Figure 11.6: Close-up of UKF Estimate of ω_3 Through One Second Measurement Dropout 136

Figure 11.7: Close-up of UKF Estimate of q_1 Through One Second Measurement Dropout 136
Figure 11.8: Close-up of UKF Estimate of q_2 Through One Second Measurement Dropout 137
Figure 11.9: Close-up of UKF Estimate of q_3 Through One Second Measurement Dropout 137
Figure 11.10: Close-up of UKF Estimate of q_4 Through One Second Measurement Dropout 138
Figure 11.11: Sample Error in State Estimate When Filter is Working Properly 145
Figure 11.12: Sample Filter Residual for One State When Filter is Working Properly 146
Figure 11.13: Sample Error in State Estimate When Filter is Working Improperly 146
Figure 11.14: Sample Filter Residual for One State When Filter is Working Improperly 147
Figure 11.15: Sample Error Angle Performance When Filter is Operating Properly 148
Figure 11.16: Sample Error Angle Performance When the Filter is Operating Improperly 148
Figure 11.17: Example of Clear Residual Departure Indicating Improper Filter Performance 149
Figure 11.18: Filter Estimate of ω_1 Corresponding to Improper Filter Performance 150
Figure 11.19: Residual Performance Following Change in Time Constant 150
Figure 11.20: Estimation of ω_1 Following Change of Time Constant 151
Figure B.1: Simulated "x" Component of Sun Vector – Inertial, Rotated, Measured 296
Figure B.2: Simulated "y" Component of Sun Vector – Inertial, Rotated, Measured 297
Figure B.3: Simulated "z" Component of Sun Vector – Inertial, Rotated, Measured 297
Figure B.4: Three Dimensional Depiction of Inertial and Rotated Sun Vector 298
Figure B.5: Simulated "x" Component of Magnetic Field Vector – Inertial, Rotated, Measured 299
Figure B.6: Simulated "y" Component of Magnetic Field Vector – Inertial, Rotated, Measured 299
Figure B.7: Simulated "z" Component of Magnetic Field Vector – Inertial, Rotated, Measured 300
Figure B.8: Three Dimensional Depiction of Inertial and Rotated Magnetic Field Vector 300
Figure B.9: Three Dimensional View of Inertial and Rotated Reference Vectors and Quaternion 301
Figure B.10: Simulated ω_1 Profile and Noisy Measurements 301
Figure B.11: Simulated ω_2 Profile and Noisy Measurements 302
Figure B.12: Simulated ω_3 Profile and Noisy Measurements 302
Figure B.13: Simulated Euler Angular Rate Profile 303
Figure B.14: Simulated Euler Angle History 303
Figure B.15: Simulated Attitude Quaternion History 304
Figure C.1: Gauss-Newton Derived q_1 Versus True q_1 305
Figure C.2: Gauss-Newton Derived q_2 Versus True q_2 306
Figure C.3: Gauss-Newton Derived q_3 Versus True q_3 306
Figure C.4: Gauss-Newton Derived q_4 Versus True q_4 307
Figure C.5: Reference Vectors Rotated Using Gauss-Newton Derived and True Quaternions 308
Figure C.6: MSE Between Vectors Rotated Using Gauss-Newton and True Quaternions 308

Figure C.7: Error in the UKF Estimate of the Rotational Rate About the “x” Body Axis 309

Figure C.8: Error in the UKF Estimate of the Rotational Rate About the “y” Body Axis 309

Figure C.9: Error in the UKF Estimate of the Rotational Rate About the “z” Body Axis 310

Figure C.10: Error in the UKF Estimate of the q_1 Component of the Quaternion 310

Figure C.11: Error in the UKF Estimate of the q_2 Component of the Quaternion 311

Figure C.12: Error in the UKF Estimate of the q_3 Component of the Quaternion 311

Figure C.13: Error in the UKF Estimate of the q_4 Component of the Quaternion 312

Figure C.14: Estimate of ω_1 Versus Measured ω_1 Versus True ω_1 312

Figure C.15: UKF Estimate of ω_2 Versus Measured ω_2 Versus True ω_2 313

Figure C.16: UKF Estimate of ω_3 Versus Measured ω_3 Versus True ω_3 313

Figure C.17: Reference Vectors Rotated Using Gauss-Newton Derived and True Quaternions 314

Figure C.18: MSE Between Vectors Rotated Using UKF derived and True Quaternions 314

Figure C.19: 3D Plot of “True” and “Estimated” Body Axes in the Inertial Frame 315

List of Tables

Table 10.1: Gauss-Newton Performance for Given Convergence Tolerance and Maximum Iterations .. 104
Table 10.2: Sample Experimental Data for Determining Optimum EKF Integration Step Size 109
Table 10.3: Sample Experimental Data for Determining Optimum Process Noise for EKF 111
Table 10.4: Sample Experimental Data for Determining Optimum EKF Time Constants 112
Table 10.5: Sample EKF Performance When no Rate Measurements are Available.............................. 114
Table 10.6: EKF Angle Measure Performance When no Rate Measurements are Available 114
Table 10.7: Sample EKF Performance When Rate Measurements are Available.................................. 115
Table 10.8: EKF Angle Measure Performance When Rate Measurements are Available 116
Table 10.9: Experimental Data for Determining Optimum UKF Integration Step Size 117
Table 10.10: Sample Experimental Data for Determining Optimum Process Noise for UKF................ 117
Table 10.11: Sample Experimental Data for Determining Optimum UKF Time Constants.................... 119
Table 10.12: UKF Performance Sensitivity to Changes in α and β Parameters...................................... 119
Table 10.13: Sample UKF Performance When no Rate Measurements are Available 121
Table 10.14: UKF Angle Measure Performance When no Rate Measurements are Available................ 121
Table 10.15: Sample UKF Performance When Rate Measurements are Available 123
Table 10.16: UKF Angle Measure Performance When Rate Measurements are Available..................... 124
Table 10.17: Summary of Algorithm Relative Performance.. 125
Table 10.18: Summary of Algorithm Relative Error Angle Performance... 127
Table 11.1: Performance at Varying Measurement Interval 129
Table 11.2: Sample Filter Performance for Sensors of Varying Accuracy 131
Table 11.3: Error Angle Comparison as a Function of Sensor Accuracy 131
Table 11.4: Sample Performance With One Second Measurement Loss 132
Table 11.5: UKF Filter Performance With Sensor Bias.. 139
Table 11.6: UKF Filter Performance During Unpredicted Motion 141
Table 11.7: Summary of Performance Before and After Change of Time Constant............................... 151
Table D.1: Sample Data From Tuning the Filters for One Simulation case.. 316

List of Appendices

Appendix A: Software Code to Support Research 164
  A.1: Attitude_filter.m 164
  A.2: Attitude_sim.m 171
  A.3: Omegagen_exp.m 186
  A.4: Omegagen_ramp.m 189
  A.5: Omegagen_fix.m 192
  A.6: Omegagen_step.m 195
  A.7: Qrot.m 199
  A.8: Gauss_Newton.m 200
  A.9: Mean_square_error.m 204
  A.10: Qframerot6.m 204
  A.11: Qvecrot3.m 206
  A.12: Orientation3.m 208
  A.13: Norates_ekf.m 210
  A.14: Norates_ukf.m 229
  A.15: Rates_ekf.m 250
  A.16: Rates_ukf.m 269
  A.17: Measadjust.m 291
Appendix B: Representative Plots Available from Data Simulation 296
Appendix C: Representative Plots Available from Filter Algorithms 305
Appendix D: Sample Results Used to Determine Process Noise and Time Constants 316

1.0 Introduction

1.1 Background

The use of sounding rockets for scientific and engineering investigation reaches back several decades. Over those many years it has progressed from the challenge of simply getting a rocket off the ground to the use of the advanced research platforms represented by today's sounding rockets. Parallel to the development of more reliable and higher performance rockets, has been a burgeoning need for more capable instrumentation and related analysis techniques. One area that has evolved substantially over time is the tracking of rocket payloads. This particular facet of analyzing rocket missions began with "eye-balling" the rocket to discover where it went. With the advent of higher performance rockets, this was no longer an adequate technique and more advanced optical methods, such as those based on telescopic cameras and theodolites, were developed. Once rockets were capable of traveling beyond visual range, tracking was often done using various radar methods. While radar tracking of the rocket position is still the principal method today, the use of Global Positioning System (GPS) techniques is making significant inroads [1], [2]. As a result of continuing development in this area, accurate and cost-effective methods currently exist for determining the position of a rocket and its payload.

As sounding rocket science became more sophisticated, it was no longer sufficient to track only the payload's position. It is now desirable, and in many cases imperative, to also estimate the payload's orientation, or attitude [3], [4], [5]. Most early efforts focused on deterministic methods of attitude determination, referencing the rocket's orientation to known objects or fields that could be sensed from the rocket while in flight. Since any measurement has some uncertainty or error associated with it, techniques were devised to minimize these measurement errors through the use of better sensors and ever more sophisticated data processing and filtering algorithms. In fact, parallel development of attitude determination systems for aircraft and missiles produced hardware and techniques suitable for use on sounding rockets. Unfortunately for the experimenter on a limited budget, many of these are based on Inertial Navigation Systems (INS) that require very accurate components and have a corresponding high cost. Following system development for aircraft and missiles, many attitude determination techniques have been devised for spacecraft. The vast majority of these are again based on INS, star trackers, and other expensive components. Here again, these efforts have focused on high performance at relatively high cost due to the overall budget and operational considerations for these systems. While much of the hardware can be, and has been, adapted to sounding rocket use, it is cost prohibitive for many experimenters.

The use of sounding rockets as scientific platforms is driven by a number of considerations. For many experiments, there is no reasonable alternative. High-altitude research balloons and aircraft cannot reach sufficient altitude, while low-Earth orbit satellites are still too high. This is the case for many ionospheric and upper-atmosphere experiments, for instance. For missions where an alternative platform does exist, cost is often the driving factor for choosing sounding rockets. In a period of tight budgets for both government and university research programs, experimenters are looking for the most cost-effective method to accomplish good science. Sounding rockets often fulfill this requirement, if they can provide adequate support for the science at an affordable price. As discussed above, affordable solutions are already available in terms of tracking rocket position. What is often lacking is affordable support in terms of accurately determining the payload's spatial orientation, or attitude.

This dissertation examines past approaches to this problem, available sensors for this application, relevant sensor fusion techniques, and promising filtering approaches. The core objective is to leverage sensor fusion and filtering techniques to design and implement an algorithm that allows for accurate, cost-effective attitude determination using sounding rocket class sensors.

1.2 Potential Applications

The primary focus of this effort is the sounding rocket attitude determination problem. There is ample evidence of the shortcomings in this particular area. The NASA Goddard Space Flight Center (GSFC) Wallops Flight Facility (WFF) is responsible for the NASA sounding rocket program. While they no longer carry out the operational program, they monitor the on-site contractor, do special analyses, support range safety, etc. The Guidance, Navigation, and Control Systems Engineering Branch at WFF has identified "assembling various 'low quality' attitude sources from sounding rocket class attitude sensors to produce a better solution to provide to the scientist" as a research interest [6]. The current solution for experimenters flying missions with WFF is one of several contractor-provided INS-based systems that must be integrated into the rocket payload. These expensive systems are essentially "rented" and then returned to the contractor for post-flight refurbishment [7]. Another current indicator of a valid requirement for improved low-cost attitude determination is the Horizontal E-region Experiment (HEX) mission launched in the spring of 2003. Pre-launch program reviews of this joint NASA/United States Geophysical Institute/University of Alaska Fairbanks project highlighted the difficulty of the attitude determination problem [8]. It is possible that the data processing algorithm proposed in this thesis, in combination with the sensors available for that mission, might have allowed for adequate accuracy without the expensive gyro-based system eventually used. At the very least, this approach could have provided experimenters a quick-look assessment of rocket payload attitude while the more accurate data from the contractor-provided system was being analyzed. In addition to these specific program needs, there is a more general impetus for this work. A number of universities currently support sounding rocket programs to conduct both instruction in engineering design and scientific investigation [9], [10]. The university programs are not typically funded at the level necessary to incorporate expensive attitude determination systems into their missions. As a result, the type and quality of possible research is limited. Development of a cost-effective system would allow not only attitude determination, but will also serve as a stepping stone to adding attitude or pointing control capabilities. Even for non-university programs that enjoy more robust budgets, an accurate low-cost system is desirable because it frees up budget dollars that can be spent elsewhere.

While sounding rockets are clearly the focus here, the algorithm designed for this effort is widely applicable. Certain provisions are made to enable the system to handle the high-dynamic case of rocket flight, but there is no reason that the algorithm cannot be applied to lower dynamic applications such as submersibles, land-based vehicles, aircraft, etc. Any and all of these can benefit from low-cost, robust and accurate attitude determination.

1.3 Research Questions and Contributions of This Work

This thesis will examine the following research topics:

- Evaluate prominent methods for representing attitude in conjunction with sounding rocket attitude filter design;
- Evaluate likely sensor fusion and filtering techniques for use in sounding rocket attitude filter design;
- Leverage the most suitable filtering and attitude representation methods to design a filter that estimates rocket attitude based on information provided by a minimal low-cost attitude sensor set;
- Demonstrate filter performance using simulated data with an emphasis on capability relative to reduced sensor cost.

The chapters that follow review other work related to this topic and detail the approach to investigating the research questions noted above. It is determined that a quaternion formulation of the attitude representing the relationship between a body fixed frame and the Earth-Centered-Inertial frame is the most useful representation. A minimal sensor suite composed of Sun sensors, magnetometers, and rate sensors is defined and Gauss-Newton error minimization provides the initial data fusion for combining the measured reference vectors into an attitude quaternion. As a result, the measurements input to the filter portion of the algorithm are reduced in number from nine to seven. The seven measurements are three body rates and four quaternion components. Four filtering algorithms are evaluated by using each to process identical simulated data sets. The four algorithms are an Extended Kalman Filter (EKF) operating on rate information derived by differencing quaternions, an EKF that incorporates rate measurements from rate sensors, an unscented Kalman filter (UKF) operating on rate information derived by differencing quaternions, and a UKF incorporating rate measurements from rate sensors. The UKF incorporating rate measurements is shown to outperform the others and is demonstrated to be capable of successfully estimating the attitude state vector under a wide range of conditions.

This work represents the first known application of the recently developed UKF to the rocket attitude determination problem. It demonstrates side-by-side comparison of EKF and UKF performance under identical experimental conditions and develops a user-friendly test bed for further evaluation of predicted filter performance under numerous conditions. It demonstrates the successful, albeit at degraded accuracy, estimation of rocket attitude motion using derived rate information in place of rates measured by rate gyroscopes or other means. A method is demonstrated for detecting anomalous filter operation in real world applications. Finally, it provides a viable alternative to the traditional, expensive, attitude determination methods by incorporating an approach suitable for implementation using low cost sensors.

2.0 Previous Work

The problem of state estimation has been addressed by many researchers using a wide variety of methods. What makes this effort unique is its focus on a solution for "low cost" systems using less than optimum sensors. The application of interest, sounding rockets, also tailors the research and guides the selection of sensors, data fusion method, and filtering approach. The following two sections provide a brief survey of the work to date that is most closely related to the goals for this research.

2.1 Solutions for Low-Cost Applications in General

The work most closely aligned with this effort is reported in [11] and [12]. Here the authors investigate the filtering of attitude data based on a low-dynamics application. Their work describes a quaternion-based filter that relies on error minimization to convert sensor measurements into a quaternion parameterization of attitude. The sensor suite evaluated is composed of magnetometers, angular rate sensors, and accelerometers combined into a single package sensor known as a MARG sensor, where MARG stands for Magnetic, Angular Rate, and Gravity. The data filtering algorithm is based upon an extended Kalman filter approach. While the filter incorporates some simplifying assumptions, it is essentially a traditional extended Kalman filter design. Their goal, like one of the goals here, is real-time estimation of orientation. The application of interest, however, is body motion simulation for insertion into virtual reality scenarios. The algorithm is demonstrated to successfully track orientation changes at rates on the order of 0.05 rad/sec, meant to simulate human body motion, but much lower than those encountered during a rocket flight.

In [13], quaternions are again used as the desired parameterization of attitude, and an algorithm is designed to estimate orientation from gravity and magnetic field measurements. Error minimization techniques are used to reduce the number of observations presented to the filter and the algorithm is based on a complementary filter instead of a Kalman filter, in an attempt to avoid the tuning required for Kalman type filters. The work described concentrates on avoiding singularities and reducing the computational burden through a "reduced order" approach. While the successful convergence of the error minimization is reported, the estimation algorithm is discussed from a theoretical perspective without results.

The authors in [14] report on the design of another quaternion-based attitude determination system that focuses on low cost sensors. In this work, magnetic field and gravity are again the two measured quantities, supplemented by acceleration derived from Global Positioning System (GPS) measurements. An iterated least squares approach is used in an error minimization routine to estimate an error quaternion and an extended Kalman filter is the basis for the filtering approach. In this algorithm, the error quaternion is estimated instead of estimating the attitude directly. The application of interest is aircraft attitude determination, and the focus is on eliminating the need for rate measurements, specifically gyroscopes. By processing both simulated and actual aircraft data, they demonstrate successful convergence of the error minimization routine and that the algorithm is able to successfully track aircraft orientation.

Bachmann et al. details yet another orientation filter based on quaternions, only in this case the filtering algorithm is based on the "complementary" filter approach and uses magnetic field, gravity measurements, and angular rate measurements as its inputs [15]. These measurements come from MARG sensors which were described earlier, and the application of interest is estimation of body posture based on limb orientation. A Gauss-Newton error minimization technique is used to reduce the number of sensor "measurements" fed into the filter. The authors report initial testing of the algorithm demonstrated successful orientation tracking at rates from 10 to 30 deg/sec.

While each of these efforts incorporates tools that may be useful for estimating the attitude of a sounding rocket, each relies on conditions that are violated by the high-dynamic nature of a rocket's operating environment. Of the works described, the application with an environment most closely matched to that envisioned in this work is the estimation of aircraft attitude. In this case, GPS measurements were differenced to get a separate acceleration measurement with which to determine what component of the accelerometer measurements were due to aircraft motion and what part were due to gravity. Such a GPS augmentation would be difficult to implement on a rocket that experiences very high linear acceleration and rotational rates. Given the limitations of these approaches relative to the sounding rocket application, the next section describes work that has been done with respect to rockets in particular.

2.2 Work on Rockets

It is difficult to find work that has been tailored specifically to the rocket attitude determination problem. The works of Zarchan [16], [17], Musoff [17], and others address the position determination/estimation problem to significant depth, but most work on attitude determination of rockets has focused on deterministic methods, or how to best determine the attitude at a given time step. The compilation by Wertz [18] and Smith's [19] discussion of methods for spinning spacecraft are exceptional references for a general overview of the attitude determination problem. References [3] and [4] present deterministic methods for determining sounding rocket attitude given measurements from a Sun sensor and a magnetometer at a given time step, but no filtering is employed to improve the estimate. Lai addresses a unique case in which magnetic field was measured in only two axes and additional measurements were provided by a Moon sensor [20]. Here again, no filtering was employed. Koehler et al. describe a deterministic approach that uses telemetry signal strength in conjunction with magnetic field aspect data [5].

Challa and Natanson [21], Crassidis and Markley [22], Murrell [23], and many others have designed Kalman-based filters to estimate attitude of spacecraft, but most of these efforts involve sensor suites not well-suited to rocket missions, and all of them estimate motion in a very different operating environment than that expected for a sounding rocket. Challa and Natanson describe a system using only Kalman filtered magnetometer data for spacecraft attitude determination, but note that the magnetic field must change rapidly enough to calculate a time derivative [21]. This would not be the case for most sounding rocket flights. The algorithm also has very long convergence times which would not be acceptable. Crassidis and Markley describe a method capable of attitude estimation without rate measurements based on Kalman filtering and error minimization, but it is tailored to the three-axis stabilized satellite case [22]. The method in [23] processes data from an inertial reference unit and star trackers, exemplifying the many applications using sensors beyond the reach of a "low cost" program. There are numerous such efforts, along the lines of those mentioned in this brief review, that are described in the literature and [24] gives a very thorough synopsis of Kalman filtering as applied to spacecraft attitude estimation. As was the case for the methods described in the last section, many of these works embody tools that may be applied to the estimation of sounding rocket attitude, but none lends itself directly to this somewhat unique application.

Based on this review of work to date, it is evident that there is a niche to be explored relative to estimating rocket attitude using low-cost sensors. Having reviewed the most pertinent work to date, the next chapter begins to lay out the tools necessary to describe, measure, and estimate the attitude of a rigid body undergoing rotational motion. Chapter 3 discusses suitable coordinate frames, Chapter 4 describes what parameters best define the attitude, Chapter 5 reviews rotational motion and how it is best modeled for a low cost estimation routine, and Chapter 6 builds a case for what sensors are best suited for measuring rocket attitude. The remaining chapters discuss data fusion approaches, filtering techniques, algorithm design, and algorithm performance.

3.0 Suitable Coordinate Axes

3.1 Introduction

In all dynamics work it is crucial to use well-defined coordinate frames. The motion of interest is referenced to these frames and a well-chosen frame will often simplify the calculations involved as well as the physical understanding of the problem. There are a number of frames that are considered most suitable for certain classes of problems. In their discussion of the spacecraft attitude determination problem, Fortescue and Stark point out that "attitude" and "orientation" are usually easily understood concepts in terms of frames.

> There must be some datum frame of reference, and once this has been chosen then the attitude of a spacecraft refers to its angular departure from this datum. A right-handed set of axes is normally used in order to define a frame of reference, and if both a datum set and a set of spacecraft axes are chosen, then the attitude may be defined in a way that may be quantified [25].

The next sections will review the common coordinate axes and examine which frames are best-suited to the sounding rocket attitude determination problem.

3.2 Common Coordinate Frames

In attitude dynamics work, there are a number of "typical" frames encountered. First, and fundamentally, there is defined an inertial reference frame. This is important because it is only in an inertial, or non-accelerating reference frame, that Newton's laws of motion are approximately correct [26]. Most often this "inertial" frame is considered to be one referenced to the "unmoving" stars. As is commonly known today, the stars are indeed moving and there most likely is no truly inertial frame. Most problems are therefore worked relative to a "sufficiently inertial" reference frame. As Kaplan points out, "practical situations dictate only that the inertial frame be a reference coordinate set which guarantees required accuracy over the time interval of interest [27]." In other words, even though all real frames are indeed moving and therefore not inertial, for practical problems, it is sufficient to select a coordinate frame which is not accelerating enough to disturb the problem solution beyond desired accuracy over the time frame of interest. For a low-dynamics problem, this sufficiently inertial frame might be a "local-level" frame, tied to some nearby geographic feature and having coordinate directions of, perhaps, north-east-down to create a right-handed frame. This might be suitable for something like a ground vehicle or a submersible. For a vehicle with higher dynamics, such as an aircraft, this may still be suitable if the motion is constrained to a local area and a relatively short duration. If the motion will continue for an extended period, however, and over great distance or at high altitude, an Earth-centered coordinate system might be more appropriate. Finally, if we are analyzing the motion of a spacecraft or rocket, then a coordinate frame referenced to the Sun or stars might be most applicable. Again, what is "sufficiently" inertial will depend on the problem at hand, and as Kaplan described, the true test is whether the chosen frame accelerates to such a degree that the desired accuracy of the solution is impacted.

Clearly, since it is the orientation of a particular object that is to be analyzed, another frame of interest may be envisioned tied to the body, often called the "body frame." This coordinate frame typically has its origin at the vehicle center of mass and is a convenient coordinate system for developing the equations of motion of a vehicle [26]. A familiar aircraft example, described by Siouris, is to choose the body x-axis pointing along the aircraft's longitudinal axis (the roll axis), the body y-axis out the right wing (the pitch axis), and the body z-axis pointing down (the yaw axis) [26]. The alignment of the frame axes with axes of symmetry of the object or with its principle axes serves to simplify the description of orientation or motion. This frame allows for the orientation of the object, body, or vehicle to be described relative to the sufficiently inertial frame in which Newton's laws of motion may be applied.

A third type of coordinate frame that is frequently encountered in attitude work, is one centered at and tied to a particular sensor. In some situations, it is convenient to define such a coordinate frame to simplify interpretation and manipulation of sensor data. There may be such a frame defined for each sensor and these frames may then be related to the overall "body" coordinate frame through a coordinate transformation. This body frame is then related, through another coordinate transformation, to the "inertial" frame. It is this last coordinate transformation that defines the body's "attitude" or spatial orientation. It is the determination of this transformation relating the body frame to the inertial frame that is the goal of this research effort.

3.3 Frames Best Suited to this Problem: Navigation Frame and Body Frame

Since the "sensor frames" for a particular application may or may not differ from the body frame and will be vehicle specific, this work concentrates on the two coordinate frames that are necessary in every case. A sufficiently inertial reference frame and a body fixed frame. As mentioned earlier, there are a number of candidates for an "inertial" frame. The most obvious candidates are a local-level frame, an Earth-Centered-Earth-Fixed (ECEF) frame, an Earth-Centered-Inertial (ECI) frame, and possibly a Sun-Centered-Inertial frame. Sounding rockets are highly dynamic vehicles that may remain aloft for substantial periods and cover large distances. This profile helps eliminate the first two candidates as "sufficiently" inertial. While none of the candidates is truly inertial, it is clear that the local level frame and the ECEF frame will move significantly during a typical flight. If a typical flight is assumed to last even 30 minutes including powered flight, coast phase and time under parachute, the Earth will rotate approximately seven and a half degrees on its spin axis. In the case of a local level frame, perhaps with its origin fixed at the launch site on or near the surface of the Earth, the origin is describing an arc at a tangential velocity approximately equal to

$$V_{tang} = R_{\oplus}\,\omega \cos \lambda \tag{3.1}$$

where

V_{tang} = tangential velocity of launch site traveling west to east (m/sec),

$R_{\oplus}$ = radius of the Earth (m),

ω = rotational rate of the Earth (rad/sec),

λ = geographic latitude of the launch site (deg).

To a rough approximation, this calculation yields an eastward velocity of the Earth's surface at the equator of about 460 meters per second. Even at high latitudes, such as at Poker Flat research range in Alaska at latitude of approximately 65 degrees North, this eastward velocity is in excess of 200 meters per second. Clearly this moving, and accelerating, origin is not "sufficiently inertial." The second candidate, the ECEF frame, suffers by a similar analysis. While the velocity of the origin, tied to the center of the Earth in its path around the Sun, is not likely to impact solution accuracy, the fact that the frame is fixed to the Earth and rotates with it means that it too will rotate through approximately seven and a half degrees during a typical 30 minute flight. This, too, is a significant change in the frame and as a result the ECEF frame is not suitable for a rocket dynamics problem. The final two candidates are far more "fixed" than the first two. The ECI frame has its origin fixed to the center of the Earth and its first axis pointed to the "first point of Aries," a "fixed" star, at the time of the vernal equinox for a particular time epoch. This axis lies in the Earth's equatorial plane. The second axis is also in the equatorial plane and is 90 degrees eastward from the first. The third axis points to the Earth's north pole and completes the right-handed system. During a typical 30 minute flight, the movement of the star reference is virtually imperceptible, and to a rough approximation, the Earth-centered origin will move on the order of .0004 radians around the Sun. As pointed out by Herbert, Wertz and others, this coordinate system is not truly inertial due to the slight translation and rotation of the frame, but is typically deemed to be sufficiently inertial for spacecraft and rocket orbital and attitude work [4], [18], [26]. The remaining candidate is similar to the ECI frame, except that its origin is at the center of the Sun and the third axis is normal to the plane of the ecliptic instead of the Earth's equatorial plane. While this frame is even more inertial than the ECI frame, for spacecraft and rocket work near the Earth, the increased accuracy of the solution does not outweigh the increased complexity of the calculations involved. Also, since many of the reference fields that are detected by attitude sensors are more easily defined in the ECI frame, this is a more convenient choice. The ECI frame that will be used as the "sufficiently inertial" frame is depicted in Figure 3.1 below.

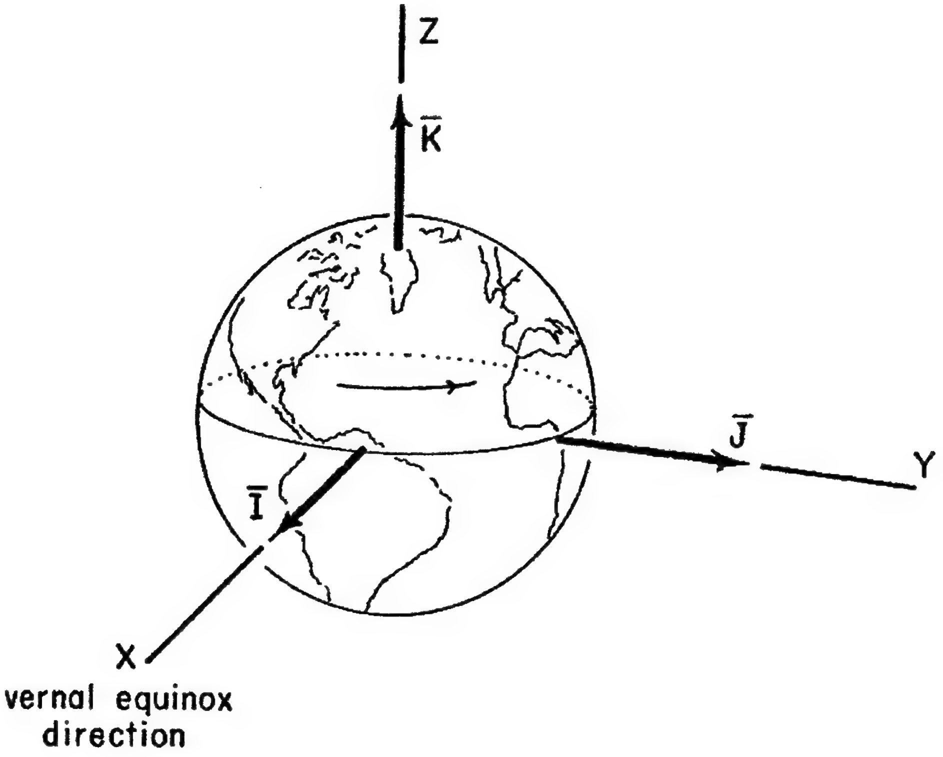


Figure 3.1: Earth-Centered-Inertial Coordinate Frame (After Figure 2.2-2 in [28])

Since attitude will be defined as a transformation relating the orientation of the body relative to the inertial frame, it is necessary to define a body-fixed frame as described earlier. While this frame may be defined anywhere within the object of interest, a well thought out choice simplifies the calculations and physical understanding of the motion. The origin is typically defined to be at the object's center of mass. For a rocket problem, there is typically a spin axis coinciding with the longitudinal axis of the vehicle. This is a natural choice for one of the body-fixed axes. The other two axes are normal to this one in order to define an orthogonal, right-handed system. While choice of the second direction is arbitrary, as long as it satisfies the orthogonality constraint, it is often chosen to align with some sensor view direction or another physical attribute of the vehicle (i.e., fin # 1, etc.). As this body frame will be unique to a particular application, it will be treated in a general sense for this research effort. Historically, several choices have been common. Originally, some researchers chose the "x" axis to correspond with the longitudinal axis [29]. A more contemporary example, the space shuttle, also designates the "x" axis to be out the nose, the "y" axis out the right wing and the "z" axis out pointed opposite the cargo bay, or in the "down" direction. In works dealing primarily with spinning objects, as many rockets and satellites are designed to do, the spin axis has often been designated as the "z" axis [30], [31], [32]. Again, this choice is somewhat arbitrary, depending on the problem. For this research effort, the body frame will be defined as a right-handed orthogonal reference frame with its origin at the vehicle center of mass and with the third, or "z" axis, aligned with the principle longitudinal axis of the rocket, positive direction out the nose. A representative body-fixed coordinate frame is illustrated in Figure 3.2 below.

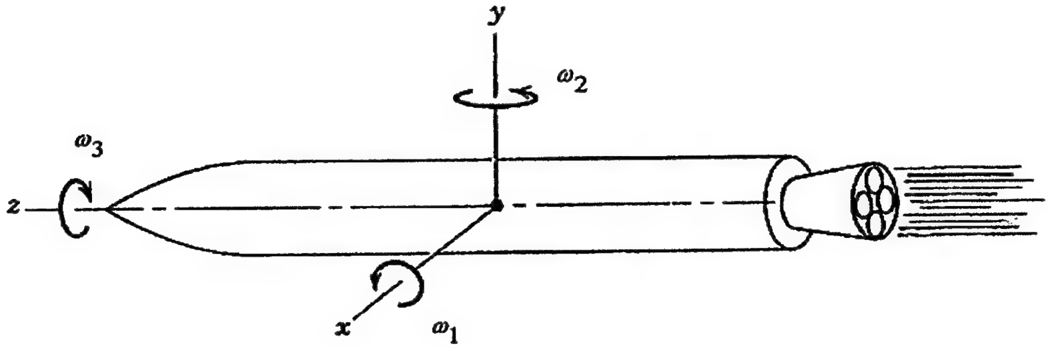


Figure 3.2: Representative Body-Fixed Coordinate Frame (After Figure 7.9-1 in [30])

Sensor axes will not be defined for this effort. It is assumed that the transformations to relate the sensor axes to the body coordinate frame are perfectly known and that no additional error is incurred as a result of inaccuracies in these transformations. Obviously, a physical vehicle will have errors associated with these transformations due to manufacturing and mounting tolerances, etc. These errors, although hopefully small, would have to be included in an overall error budget.

Devising an algorithm to determine rocket attitude involves not only a definition of coordinate frames, but also a selection of how to parameterize the attitude and thereby the transformation between the frames. There are numerous approaches to this, with a few being relatively common. Each has its advantages and disadvantages, and the next sections will examine the more common methods and select the one best-suited to the sounding rocket attitude determination problem.

4.0 Coordinate Parameterizations

4.1 Introduction

As laid out in the last chapter, the minimum set of frames required to analyze attitude or orientation includes a "sufficiently inertial" frame in which Newton's laws approximately hold true, and a body frame fixed to the object. Since attitude motions are normally represented in the spacecraft body fixed frame, which is rotating and accelerating, the subject of relative motion and transformation of coordinates plays an important role in attitude dynamics [33]. Historically, spatial attitude descriptions have been accomplished using a number of different coordinate parameterizations [26], [34], [35], [36], [37], [38]. The following paragraphs discuss the parameters most commonly appearing in the literature, their advantages and disadvantages relative to this application, and the choice of parameterization for this research.

4.2 Common Coordinate Parameterizations

The definition of an object's attitude may be distilled down to the description of the transformation between the body-fixed frame and the datum frame [34]. This transformation may be described by one of many coordinate parameterizations. At the root of these parameterizations, attitude can be viewed from either a rectangular or a spherical coordinate perspective. While spherical coordinates often help when visualizing the physical motion of an object, their manipulation involves the heavy use of trigonometric functions, which are computationally more intensive from a computing point of view [18]. In general, the availability of faster computer chips has made this less of an issue. However, since the focus of this work is on low cost systems, available processing power may be limited. In light of the desire to implement the algorithm in real time, the computational savings associated with using rectangular coordinates make their use worthwhile. Even after narrowing the approach to one based on rectangular coordinates, there are a large number of coordinate parameterizations that have been studied and used for attitude determination work [35]. From this large group, a few stand out as the most suitable for the sounding rocket problem.

The parameterization considered by some to be the "fundamental" representation is the familiar direction cosine matrix [37]. This matrix maps vectors in the datum frame to the body frame, or vice versa, by employing a set of three direction cosine elements to determine the orientation of each inertial axis with respect to the three axes of the body frame [26]." Even though direction cosines can be considered the "standard" by which other representations might be judged, Markley points out that other parameterizations might have significant advantages in certain applications [37]. One disadvantage of the direction cosine matrix is that its 3x3 matrix structure requires nine elements. Since an orientation or rotation of a rigid body possesses only three degrees of freedom, this method requires six constraints to get from nine parameters to the minimum of three that are theoretically required. While this method is relatively easy to understand and well adapted to computer programming [3], in the end the direction cosine method requires the manipulation of expensive trigonometric functions and imposes a computer storage burden that is not required when using some alternative methods.

While direction cosines may be the best known method for describing orientation, a second often-used method relies on Euler angles [12]. Since any rotation of a rigid body may be completely described using only three parameters [37], it is very appealing to choose a method that uses this minimum set. The Euler angle approach is perhaps the best known such method. Devised by Euler, this theorem states that "*Any two independent orthonormal coordinate frames can be related by a sequence of rotations (not more than three) about coordinate axes, where no two successive rotations may be about the same axis* [38]." These three rotations made in a precise order about specific axes in order to obtain an orientation are the Euler angles. There are multiple combinations of angles and axes that yield the same final orientation, so in this sense the "Euler angles" are not unique [26], [39]. One common representation, a 3-1-3 sequence, is shown in Figure 4.1 on the next page. In this sequence, the first rotation is through angle ψ about the original "third" axis, the second rotation is of magnitude θ about the new "first" axis, and the final rotation is through φ about the new "third" axis. In this figure, the upper-case "E" vectors are the inertial directions, while the lower-case "e" vectors are those attached to the body frame. Tying this representation back the direction cosines discussed earlier, the direction cosine matrix (DCM) that transforms the inertial vectors into the rotated frame, in terms of the Euler angles defined here, is [40]

$$DCM = \begin{bmatrix} \cos\phi\cos\psi - \sin\phi\cos\theta\sin\psi & \cos\phi\cos\psi + \sin\phi\cos\theta\cos\psi & \sin\phi\sin\theta \\ -\sin\phi\cos\psi - \cos\phi\cos\theta\sin\psi & -\sin\phi\sin\psi + \cos\phi\cos\theta\sin\psi & \cos\psi\sin\theta \\ \sin\theta\sin\psi & -\sin\theta\cos\psi & \cos\theta \end{bmatrix}. \tag{4.1}$$

As is well-documented in the literature, however, the large computational burden associated with the use of this three parameter representation, as well as the fact that for some rotations the angles become undefined, often makes them unsuitable or undesirable for real-world problems [15], [25], [36], [35], [38], [41]. This problem with singularities is often referred to as the "gimbal lock" problem when working with gimbaled inertial navigation systems, but also causes computational difficulties in mathematical algorithms [26]. For example, with the 3-1-3 sequence defined here, it is clear that if rotation of the rocket causes the body frame "third" axis to become aligned with the reference frame "third" axis, then the angle φ becomes undefined, as there is no longer a unique line of intersection between the planes defined by the "first" and "second" axes of each frame. At that moment, and for as long as the two "third" axes remain aligned, the ability to distinguish one of the three degrees of freedom is lost. In fact, all known three-parameter methods display singularity problems at certain orientations [26]. Since a highly dynamic sounding rocket can be expected to transition through many possible orientations, an Euler angle-based algorithm is likely to encounter such difficulties [34]. Therefore Euler angles, or other three parameter approaches, are ill-suited parameterizations for this problem [13].

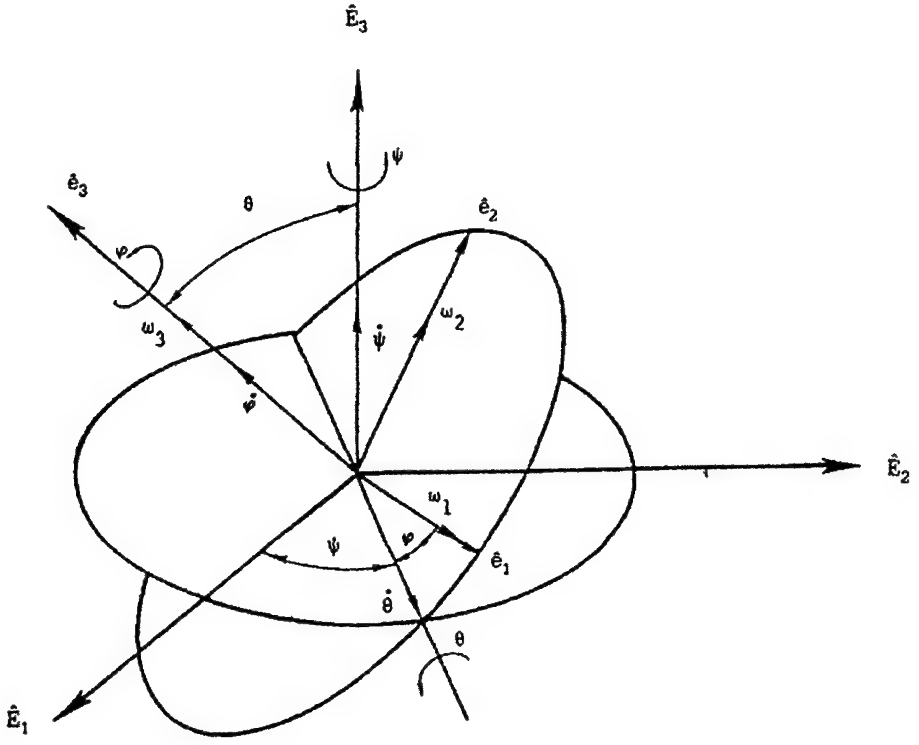


Figure 4.1: Classic Euler Rotations of a Rigid Body (After Figure 1.8 in [40])

4.3 Parameterization Best Suited to This Problem: Quaternions

A number of four parameter approaches have been devised to avoid the singularity issues inherent in three parameter methods, where one or more parameters become undefined at certain orientations. The basis for these methods is a set of parameters known as quaternions [26]. Quaternions are alternately known as Euler symmetric parameters [37], or less commonly as Cayley-Klein parameters [34]. In the words of Siouris, "a quaternion is a means of describing angular orientation with four parameters, the minimum redundancy that removes the indeterminate points of three-parameter (Euler angles) descriptions [26]." The foundation for the quaternion approach is Euler's theorem which states that any finite rotation of a rigid body can be expressed as a rotation through some angle about some fixed axis [40], [42]. In one interpretation, three of the four quaternions define the fixed rotation axis while the fourth gives the magnitude of the angle of rotation about that axis [26], [34], [38]. In this manner, the four quaternions unambiguously define the transformation. The price for this lack of ambiguity is extra complexity in the form of one redundant parameter, which in turn requires one constraint. Another drawback is that quaternions present no obvious physical interpretation of the rotation [40]. The benefits are the absence of singularities, an efficiency of computation relative to that required for direction cosines or an Euler angle approach, and greater robustness with respect to numerical round off errors [37], [38], [40], [43].

Given these advantages associated with a quaternion parameterization, this research effort will focus on representing sounding rocket attitude using quaternions. Since these will be an integral part of the development, it is important to understand what they are and their relevant characteristics. Hamilton is credited with inventing quaternions in 1843. They are described as hyper-complex numbers of rank 4, composed of a scalar part and a three-dimensional vector part [38]. A quaternion q may be expressed as [40], [44]

$$q = \mathbf{q} + q_4 = iq_1 + jq_2 + kq_3 + q_4 . \tag{4.2}$$

where $i, j,$ and k are unit vectors in the hyper-dimensional plane. Similar to the familiar complex numbers, the conjugate of a quaternion is formed by simply taking the negative of the imaginary part

$$q^* = -\mathbf{q} + q_4 = -iq_1 - jq_2 - kq_3 + q_4 . \tag{4.3}$$

Depending on the author, the index for the elements differs. For instance, Kuipers and Siouris place the scalar part first in their notation and use the index zero [26], [38],

$$q = q_0 + \mathbf{q} = q_0 + iq_1 + jq_2 + kq_3 \tag{4.4}$$

and correspondingly, the conjugate is

$$q^* = q_0 - \mathbf{q} = q_0 - iq_1 - jq_2 - kq_3 . \tag{4.5}$$

Regardless of the ordering and indexing of the elements, these notations represent equivalent concepts. This variation in notation causes no real problems, but when using expressions for the time derivative of the quaternion, etc., from the literature, care must be taken to be cognizant of the notation upon which they are based.

In addition to inventing the quaternion itself, Hamilton also developed the now well-established rule

$$\begin{aligned} i^2 = j^2 = k^2 = ijk = -1 \\ ij = -ji = k \\ jk = -kj = i \\ ki = -ik = j \end{aligned} \tag{4.6}$$

for dealing with the operations on the vector part of the quaternion [38], [44]. Theoretically there exist hyper-complex numbers of even higher rank, but as pointed out by Kuipers, few applications have been found for them. For the three-dimensional orientation problem, numbers of rank 4, known as quaternions are well suited. It is now generally acknowledged that the principle use for quaternions is as a rotation operator to accomplish three-dimensional rotations [38]. Indeed, given the structure defined in equations (4.2) and (4.4) above, it is clear that traditional three-dimensional vectors may be represented as quaternions with a zero scalar part.

Having established the above definition of quaternions, it is helpful to understand some of their relevant properties. Many authors' works on dynamics and attitude determination have included at least a summary of quaternions [26], [37], [40], [43] and Kuipers has written an entire text on the quaternion as a rotation operator [38]. Here I will summarize the pertinent results that will be used in the later development of the attitude determination algorithm. A more rigorous development may be found in one of the works referenced above.

Mathematically, the behavior of quaternions is similar in many ways to the familiar complex numbers. Perhaps the greatest difference, and a restriction always to be mindful of, is that quaternions are generally not commutative under multiplication as is evident from equation (4.6) above. Intuitively this makes sense, given that quaternions are used as rotation operators and rotations must be done in a specific order. In fact, except for certain special cases, changing the order of a sequence of rotations changes the end result. Additionally, a quaternion can be represented by a rotation matrix, which is not commutative. So, while this result is not surprising, it is one to remain aware of while manipulating quaternions. Rotation matrices representing *incremental rotations* are one special case that is an exception to this rule. These would be rotations with an angle of rotation, θ, such that we may consider $\sin\theta \approx \theta$ and $\cos\theta \approx 1$. For these incremental rotations, the rotation matrices do indeed commute under multiplication [38]. Similarly, as is easily demonstrated, the quaternions for this special case also commute under multiplication.

As stated earlier, the use of quaternions to represent rotations is based on Euler's idea that any rotation can be defined as a vector about which the rotation occurs and an angle defining the magnitude of the rotation. While the detailed development is beyond the scope of this work, the transformation of a vector **u** that corresponds to multiplication by rotation matrix A,

$$\mathbf{u}' = A\mathbf{u}, \tag{4.7}$$

is accomplished using quaternion algebra as

$$\mathbf{u}' = q^{*}\mathbf{u}q \tag{4.8}$$

where q^* is the conjugate of q [44]. In a matrix format, this quaternion representation can be written as [38]

$$\mathbf{u}' = Q\mathbf{u} \tag{4.9}$$

which when expanded yields

$$\begin{bmatrix} u_1' \\ u_2' \\ u_3' \end{bmatrix} = \begin{bmatrix} (2q_4^2 - 1 + 2q_1^2) & (2q_1q_2 + 2q_4q_3) & (2q_1q_3 - 2q_4q_2) \\ (2q_1q_2 - 2q_4q_3) & (2q_4^2 - 1 + 2q_2^2) & (2q_2q_3 + 2q_4q_1) \\ (2q_1q_3 + 2q_4q_2) & (2q_2q_3 - 2q_4q_1) & (2q_4^2 - 1 + 2q_3^2) \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \tag{4.10}$$

or alternatively, using a different grouping of terms, in the form [40]

$$\begin{bmatrix} u_1' \\ u_2' \\ u_3' \end{bmatrix} = \begin{bmatrix} (q_1^2 - q_2^2 - q_3^2 + q_4^2) & 2(q_1q_2 + q_3q_4) & 2(q_1q_3 - q_2q_4) \\ 2(q_1q_2 - q_3q_4) & (-q_1^2 + q_2^2 - q_3^2 + q_4^2) & 2(q_1q_4 + q_2q_3) \\ 2(q_1q_3 + q_2q_4) & 2(-q_1q_4 + q_2q_3) & (-q_1^2 - q_2^2 + q_3^2 + q_4^2) \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}. \tag{4.11}$$

It should be noted that this operator is interpreted as a *frame* rotation, representing the same vector in the rotated frame. The corresponding *vector* rotation operator, which represents the rotated vector in the original frame, is simply [38]

$$\mathbf{u}' = q\mathbf{u}q^*. \tag{4.12}$$

Since rotation matrices are orthogonal, the corresponding matrix for this quaternion operator is simply the transpose of those shown above, or Q^T. As noted by Kuipers, "A frame rotation through a certain angle is entirely equivalent to a point rotation (about the same axis) but through the *negative* of that angle [38]."

Viewing the quaternion as a rotation operator, a convenient representation for a quaternion, q, is [38]

$$q = \cos\frac{\alpha}{2} + \mathbf{k}\sin\frac{\alpha}{2} \tag{4.13}$$

where the rotation is about an axis along the vector **k** and the magnitude of the rotation is α. From this expression it is clear that in order for q to represent a pure rotation, with no change in magnitude, **k** must be a unit vector so that the overall magnitude of q is one. Representing the unit quaternion q using the notation from equation (4.2),

$$q = iq_1 + jq_2 + kq_3 + q_4 = \mathbf{q} + q_4 \tag{4.14}$$

and setting this equal to (4.13) and rearranging, we see that the angle of rotation, α, represented by the unit quaternion q is [38]

$$\alpha = 2\arccos(q_4). \tag{4.15}$$

about an axis in the direction of **q**.

As described earlier, the use of quaternions requires one constraint since there are four parameters and only three degrees of freedom in the attitude determination problem. As noted in the preceding paragraph, when using quaternions as a rotation operator, it is necessary that the norm must be unity, and this provides the constraint. To verify that q is indeed a unit quaternion, the norm may be calculated as [40]

$$N(q)=\sqrt{q_1^2+q_2^2+q_3^2+q_4^2}\,. \tag{4.16}$$

and the constraint may therefore be viewed as

$$N(q)=\sqrt{q_1^2+q_2^2+q_3^2+q_4^2}=q_1^2+q_2^2+q_3^2+q_4^2=1\,. \tag{4.17}$$

In the case where q is indeed a unit quaternion, an additional useful property is that its inverse is simply its conjugate [38],

$$q^{-1}=q^{*}\,. \tag{4.18}$$

As alluded to earlier, one principle advantage of quaternions over rotation matrices is an economy of computation when doing successive rotations. For the case of two rotations of a three-dimensional vector, the individual 3×3 rotation matrices may be multiplied together to obtain the matrix for the combined rotation. This requires 27 multiplications. In the case of quaternions operating on a vector **u**, in accordance with equation (4.8) above,

> "suppose that p and q are unit quaternions which define the quaternion rotation operators
>
> $$L_p(\mathbf{u}) = p^{*}\mathbf{u}p \text{ and } L_q(\mathbf{u})=q^{*}\mathbf{u}q\,. \tag{4.19}$$
>
> Then the quaternion product pq defines a quaternion rotation operator L_{pq} which represents a sequence of operators, L_p followed by L_q. The axis and the angle of rotation of the composite rotation operator are those represented by the quaternion product pq [38]."

The quaternion product pq may be expressed as [38], [44]

$$pq=(p_4q_4-\mathbf{q}\cdot\mathbf{p}+p_4\mathbf{q}+q_4\mathbf{p}+\mathbf{p}\times\mathbf{q}) \tag{4.20}$$

or in its expanded form as [38], [44]

$$\begin{aligned} pq&=(-p_1q_1-p_2q_2-p_3q_3+p_4q_4)\\ &+i(p_1q_4+p_2q_3-p_3q_2+p_4q_1)\\ &+j(-p_1q_3+p_2q_4+p_3q_1+p_4q_2)\\ &+k(p_1q_2-p_2q_1+p_3q_4+p_4q_3) \end{aligned}\,. \tag{4.21}$$

In addition to having no singularities, and the advantage of being a purely algebraic operation, this results in a reduction in required multiplications from 27 to 16 for each combined rotation [38], [40]. For the problem of attitude determination, many such successive rotations are required and the result is a substantial savings in computation when using quaternions in place of rotation matrices. This further bolsters the choice of quaternions for a real-time application such as that envisioned here.

Since this research effort will use quaternions to represent the attitude of the vehicle, the quaternion rates will need to be related to the body attitude rates. Therefore, another critical construct for the development of the attitude determination algorithm is the expression for the time derivative of the quaternion. While the development of this expression can follow several approaches [26], [40], [45], Chobotov presents a convenient representation that will be used in this work [40],

$$\frac{d}{dt}(q_j) = \dot{q}_j = \frac{1}{2}\sum_{i=1}^{4} q_{ij}\omega_i . \tag{4.22}$$

Here q_{ij} are the components of the four dimensional orthogonal rotation matrix

$$q_{ij} = \begin{pmatrix} q_4 & q_3 & -q_2 & -q_1 \\ -q_3 & q_4 & q_1 & -q_2 \\ q_2 & -q_1 & q_4 & -q_3 \\ q_1 & q_2 & q_3 & q_4 \end{pmatrix} \tag{4.23}$$

and ω_i are the body rates as measured in the reference frame. Expanding equation (4.22) yields

$$\begin{pmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \\ \dot{q}_4 \end{pmatrix} = \frac{1}{2}\begin{pmatrix} q_4 & -q_3 & q_2 & q_1 \\ q_3 & q_4 & -q_1 & q_2 \\ -q_2 & q_1 & q_4 & q_3 \\ -q_1 & -q_2 & -q_3 & q_4 \end{pmatrix}\begin{pmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ 0 \end{pmatrix}. \tag{4.24}$$

Additionally, as Chobotov points out, "since the terms are linear in ω_1, ω_2, ω_3, and also in q_j, this equation can alternatively be written as [40]

$$\begin{pmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \\ \dot{q}_4 \end{pmatrix} = \frac{1}{2}\begin{pmatrix} 0 & \omega_3 & -\omega_2 & \omega_1 \\ -\omega_3 & 0 & \omega_1 & \omega_2 \\ \omega_2 & -\omega_1 & 0 & \omega_3 \\ -\omega_1 & -\omega_2 & -\omega_3 & 0 \end{pmatrix}\begin{pmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{pmatrix}." \tag{4.25}$$

Finally, it will prove necessary to relate a quaternion to a known set of Euler angles. This requires the derivation of an expression for each of the quaternion components. This may be approached from the perspective of describing each Euler angle as a rotation about the appropriate axis by the magnitude of the

Euler angle of interest. This allows each in the 3-1-3 sequence to be represented according to equation (4.11) as a rotation quaternion yielding

$$\begin{aligned} q_\psi &= \cos\frac{\psi}{2} + \hat{k}\sin\frac{\psi}{2} \\ q_\theta &= \cos\frac{\theta}{2} + \hat{\imath}\sin\frac{\theta}{2} \\ q_\phi &= \cos\frac{\phi}{2} + \hat{k}\sin\frac{\phi}{2} \end{aligned} \quad . \tag{4.26}$$

These may then be combined as three subsequent rotations using equation (4.20) or (4.21) above. After the quaternion multiplications are carried out according to the rules in equation (4.6), and terms are combined, the following relationships emerge allowing the calculation of quaternion components given the Euler angles:

$$\begin{aligned} q_1 &= \cos\frac{\psi}{2}\sin\frac{\theta}{2}\cos\frac{\phi}{2} + \sin\frac{\psi}{2}\sin\frac{\theta}{2}\sin\frac{\phi}{2} \\ q_2 &= -\cos\frac{\psi}{2}\sin\frac{\theta}{2}\sin\frac{\phi}{2} + \sin\frac{\psi}{2}\sin\frac{\theta}{2}\cos\frac{\phi}{2} \\ q_3 &= \cos\frac{\psi}{2}\cos\frac{\theta}{2}\sin\frac{\phi}{2} + \sin\frac{\psi}{2}\cos\frac{\theta}{2}\cos\frac{\phi}{2} \\ q_4 &= \cos\frac{\psi}{2}\cos\frac{\theta}{2}\cos\frac{\phi}{2} - \sin\frac{\psi}{2}\cos\frac{\theta}{2}\sin\frac{\phi}{2} \end{aligned} \quad . \tag{4.27}$$

Having summarized the advantages that make quaternions the desired parameter for this problem, and their characteristics that will be important for this development, the next logical step is an examination of the dynamics involved in the attitude motion of a sounding rocket.

5.0 Rocket Rotational Dynamics

5.1 Introduction

As laid out in the last chapter, quaternions present a preferred method for describing the orientation of an object in space. A time history of the quaternion components will provide a picture of the three-dimensional rotational motion of the object, while at each time step, the quaternion rates can be related to the rotational rates measured in the body frame. As part of the development of an accurate attitude solution, a basic understanding of the rotational dynamics of a rocket is helpful.

5.2 Basics of Rocket Rotational Motion

Even a rudimentary look at the mechanisms that drive rocket rotational motion quickly becomes complex. When looking at the entire flight regime, the launch and ascent phases are especially troublesome as they take place principally within the sensible atmosphere. This presents a number of difficulties when attempting to develop an analytical expression to describe the motion of a rocket as the result of a summation of forces and moments acting on the rocket. Large portions of major works have been devoted to such studies [29], [46], [47], and yet there is no straightforward method to predict rocket motion beyond a gross sense. A rigorous, analytical approach depends heavily on the physical characteristics of the specific vehicle and on the atmospherics present at the time of launch. In describing drag and lift, two of the most significant forces impacting rocket motion, Feodosiev and Siniarev point out that,

> The mechanism of formation of aerodynamic forces is quite complex. Determination of D and L forces by computation, even in the simplest cases, often constitutes an insoluble problem. Therefore, currently, the determination of aerodynamic forces (values of C_D and C_L) is done by approximate calculations which are used as a guide and which later on are corrected by wind tunnel tests [29].

Since their work, large six degree of freedom computer models have been developed to help predict such motion, but they are very computationally intensive and require the input of many parameters describing both the aforementioned launch conditions and vehicle parameters. Even with the availability of such a modeling program, complications exist due to imperfections in manufacture and assembly [29], [47], resulting in a difference between the actual rocket parameters and those entered into the program. Adding to the difficulty of analytically predicting motion, is the fact that the motion itself is not simple. For instance, "an error in alignment of the maximum moment of inertia with the symmetric body axis, about which the spacecraft is spun, is common and produces a coning of the spin axis about the net angular momentum vector [39]." Additionally, many sounding rocket vehicles use a combination of fin and spin stabilization. Among other things, this leads to a "coupling" of the motion in the pitch and yaw axes, especially while still in the atmosphere. As described by Feodosiev and Siniarev, "If the rocket moves with an angle of attack, then, with the appearance of the yaw angle, the rocket will roll...The right stabilizer has additional velocity, the lift force on it increasing, whereas on the left it decreases. A moment develops, rotating the rocket about its longitudinal axis and giving rise to rocket roll [29]." In addition to "cross-coupling" of motion due to aerodynamics, there is cross-coupling associated with a lack of mass symmetry combined with the large angular momentums resulting from high spin rates [25]. All of these issues contribute to a level of complexity that does not lend itself to an easily transportable algorithm that would be suitable for low-cost applications and across various different platforms.

Faced with the complexities of the true dynamics, other works [25], [31], [32], [37], [45], [48], often treat the rotational motion of rockets in a more general and simplified sense, and that is the approach that will be summarized here. These analyses apply to any spinning body, and the authors typically analyze motion resulting from moments created about the various body principle axes without delving into the details of what generates those moments. Unfortunately, it is often the details that distinguish between the theoretical motion and the true motion of the rocket.

From a very simplistic view, it would at first appear that sensors mounted within a rocket could measure the rotational rate about each axis. These rates, in turn, could then be integrated with respect to time to get a rotational angle at each time step, thereby indicating the rocket attitude. This approach is appealing as it parallels the familiar "Newton's Second Law of Motion" describing the translational motion. This principle states that the time rate of change of momentum is equal to the summation of forces applied to the object. The more well-known version being "the sum of the forces is equal to mass times acceleration," which is the result one gets by assuming a constant mass. For translational motion, the acceleration obtained by the force summation is then integrated once to get velocity, and then again to get position. The not so obvious requirement is that Newton's laws only hold true in an inertial reference frame. So, while there is a rotational corollary to the familiar Newton's Second Law, the sum of the moments is equal to the time rate of change of angular momentum, and the angular acceleration and angular rates can be integrated to get angular orientation, this also must happen with respect to an inertial reference frame. As described in Chapter 3, the body frame is not an inertial frame. Therefore, as pointed out by many authors [32], [38], [40], the rates expressed in the body frame are not suitable for integration. If, however, the rates can be determined in a sufficiently inertial frame, they can be integrated with respect to time to get rotational orientation with respect to the inertial frame. As will be shown in a later section, this will be an integral concept in the simulation phase of the work accomplished here.

In his discussion of what presents a suitable set of coordinates for integration, and therefore for predicting attitude, Meirovitch points out that

> Nor do the direction cosines l_{ij} meet this need, because they are not independent, which rules them out from qualifying as a set of generalized coordinates. To describe the orientation of a rigid body in space we need three independent coordinates. Such a set of coordinates is not necessarily unique, and...A set which enjoys wide acceptance consists of *Euler's Angles*. These are three successive angular displacements which can adequately carry out the transformation from one Cartesian system of axes to another, although the rotations are not about three orthogonal axes. Moreover, the three components of the body angular velocity can be expressed in terms of Euler's angles and their time derivatives [32].

So, to summarize, the body rates cannot be integrated, but the Euler angle rates may, since they are inertial quantities. Of course, care must be taken to account for the singularities associated with using Euler angles, as described in Chapter 4.

The overarching equation governing the rotational motion of rigid bodies is indeed Newton's Second Law of Rotational Motion...the sum of the moments is equal to the time rate of change of angular momentum. The more familiar version of this law is "the sum of the moments is equal to rotational moment of inertia times the angular acceleration." This simplification is based on the assumption that the moment of inertia does not change with time. An expansion of this law, taking into account all three axes, and substituting relations for the rates measured in the body frame, yields the familiar Euler's Equations of Rotational Motion [30]

$$\begin{aligned} M_1 &= A\dot{\omega}_1 + (C-B)\omega_2\omega_3 \\ M_2 &= B\dot{\omega}_2 + (A-C)\omega_1\omega_3 \\ M_3 &= C\dot{\omega}_3 + (B-A)\omega_1\omega_2 \end{aligned} \tag{5.1}$$

where

M_1 = moment about the body first axis,

M_2 = moment about the body second axis,

M_3 = moment about the body third axis,

ω_1 = rotational rate about the body first axis,

ω_2 = rotational rate about the body second axis,

ω_3 = rotational rate about the body third axis,

A = principle moment of inertia about the body first axis,
B = principle moment of inertia about the body second axis,
C = principle moment of inertia about the body third axis.

In turn, the body rates may be conveniently related to the inertial Euler angle rates using the "Euler angle rate equations." There is no universally accepted set of Euler angles or notation for them, but using the angle sequence depicted in Figure 4.1, and in the notation of Chobotov [40] and Kaplan [27], these equations are

$$\begin{aligned} \omega_1 &= \dot{\psi}\sin\theta\sin\phi + \dot{\theta}\cos\phi \\ \omega_2 &= \dot{\psi}\sin\theta\cos\phi - \dot{\theta}\sin\phi \\ \omega_3 &= \dot{\phi} + \dot{\psi}\cos\theta \end{aligned} \tag{5.2}$$

where

ω_1 = rotational rate about the body first axis,
ω_2 = rotational rate about the body second axis,
ω_3 = rotational rate about the body third axis,
ψ = clockwise angle about third axis,
θ = clockwise angle about "new" first axis,
ϕ = clockwise angle about "new" third axis.

Since the Euler angles do not constitute an orthogonal transformation [40], the inverse transformation relating the Euler angle rates as a function of the body rates is not simply the transpose of the transformation matrix derived from (5.2) above, but must be solved for. The resulting relationship in a matrix form is [27], [40]

$$\begin{pmatrix} \dot{\psi} \\ \dot{\theta} \\ \dot{\phi} \end{pmatrix} = \frac{1}{\sin\theta}\begin{pmatrix} \sin\phi & \cos\phi & 0 \\ -\sin\phi\cos\theta & -\cos\phi\cos\theta & \sin \\ \cos\phi\sin\theta & -\sin\phi\sin\theta & 0 \end{pmatrix}\begin{pmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{pmatrix} \tag{5.3}$$

where

ψ = clockwise angle about third axis,
θ = clockwise angle about "new" first axis,
ϕ = clockwise angle about "new" third axis,
ω_1 = rotational rate about the body first axis,

ω_2 = rotational rate about the body second axis,

ω_3 = rotational rate about the body third axis.

The relationships between the body rates and the Euler angle rates given in equations (5.2) and (5.3) above, as well as the relationship between body rates and quaternion rates given in equation (4.22) of the previous chapter, provide convenient methods to express rotational motion in the body and inertial frames. Each will be used in the development laid out in a later chapter.

5.3 Simplified Model of Rocket Motion

A number of goals associated with this effort drive a move to a simplified model of the rocket dynamics. First, it is desirable that the model be applicable to various, if not all, rockets with little re-derivation required. Additionally, in keeping with the desired "low-cost" nature of the solution, the procurement and hosting of complicated six-degree of freedom models is not realistic. Even the "simplified" dynamic equations summarized in the last section require accurate prediction of moments acting on the rocket throughout its flight, as well as accurate measurement of the physical parameters that make up the moments of inertia. While these efforts are routinely accomplished for very expensive missions, they do not lend themselves to a "low-cost" effort such as that undertaken by most universities.

First, it is important to realize where dynamic models will be required for this effort and what level of fidelity is necessary. During development of the algorithm and subsequent testing, rocket motion will be simulated to provide a "truth" reference. Noisy measurements of this truth reference will be simulated as well. The algorithm will operate on these in an attempt to recover the uncorrupted motion. For this simulation phase, the best case would be a situation where you could "fly" the rocket through a number of perturbing moments and come up with the exact expected motion of the vehicle. For the reasons discussed in the previous section, a model of this fidelity is not realistic, and for simulation purposes is not necessary. To develop and test the algorithm, motion that is typical of real motion, having similar complexity, will be simulated. Second, several of the filtering techniques to be examined require a dynamic model to propagate states forward in time. Fortunately, due to the small time steps used and the fact that the model is embedded within the filter, this model does not need to be very high fidelity, as will be demonstrated in a later section.

Given these inherent limitations and the requirements for this effort, the following simplified dynamic model is proposed. This model is used in the filtering efforts as a relatively low fidelity, but adequate, model of the rocket rotational motion. It has the benefit of requiring no measurement or derivation of rocket physical parameters, and is therefore transportable between vehicles. In fact, this simplified model should allow this algorithm to be applied to the attitude determination of any rotating object, as long as the filter is tuned to the expected motion. As presented by Marins [11], the simplest dynamic model for generating angular rates is one based on rates driven by white noise, in which the derivative of the rates is a function of the rates themselves. As described in the previous paragraph, this model is to be embedded in the filter and therefore does not need to be of high fidelity. Implicit in its use, however, is the assumption that the inertia of the rocket prevents its motion from changing radically over the duration of one filter time step. This is necessary because the model represents the new rotational rate as the previous rate perturbed by a small, random, acceleration scaled by the time constant τ_r. Longer time steps may indeed require a higher fidelity dynamic model for the propagation of states within the filter. For this application, however, at small time steps relative to the motion of the rocket, this rudimentary model is adequate. The dynamic model for the angular rates may be represented by Figure 5.1 below. In this figure, ω is a 3×1 vector of angular rates ω_1, ω_2, and ω_3 in the body frame, w_r is a 3×1 vector of white noise components generating ω_1, ω_2, and ω_3, and τ_r is a 3×1 vector of time constants corresponding to ω_1, ω_2, and ω_3. The determination of both the time step and the time constants will be elaborated upon in Chapter 10. Marins et al. [12], used this model successfully with respect to a low-dynamics submersible vehicle. It will be demonstrated in a later section that, with proper scaling through the time constants and sufficiently small time steps, it also functions adequately for the much higher rotation rates seen in the case of rocket motion.

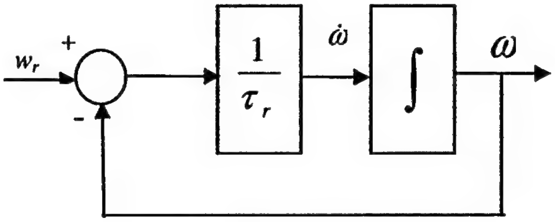


Figure 5.1: White Noise Driven Angular Rates (After Figure 3.1 in [11])

Having examined some of the characteristics that make rocket motion complex, as well as surveying the equations that help describe this motion, it is now necessary to investigate how the motion might be monitored, or measured, given the constraints of this effort. These measurements will provide the raw material from which the algorithm determines a best estimate of the rocket's rotational motion.

6.0 Attitude Sensing

6.1 Introduction

With a basic understanding of the rotational motion of a rocket and its payload, the means of measuring the motion can now be examined. While a perfect dynamic model would allow a precise projection of the motion, as pointed out in Chapter 5, such a model does not exist. Therefore, any algorithm that is designed to provide a best estimate of rocket attitude must be supplied with measurements that can be combined and filtered to yield an accurate solution. This chapter will evaluate the types of measurements available, and present those selected as the best choice for this effort.

6.2 Common Attitude Sensors

A survey of attitude sensors reveals that there are many possible sensors, but few that are commonly used in practice. From this reduced set of sensors, an even smaller set is best-suited for use on a sounding rocket platform. A summary of this survey follows.

Chobotov provides a succinct list of the most commonly used attitude sensors to include Earth horizon, Sun, star and inertial measuring units (IMUs). He goes on to point out that magnetometers and radio interferometers are also used [40]. The Global Positioning System (GPS), long known as an accurate source of position information, is also being used for attitude determination in a limited number of applications [49]. This list is based primarily on satellite attitude sensing, although each entry might also find its place on a sounding rocket mission, if sufficient budget is available. Each method/sensor has its benefits and drawbacks and some are not especially well-suited to the sounding rocket mission. Combined with the cost and facility constraints associated with the goal of this research, many can be eliminated, leaving a few that present themselves as likely candidates.

Earth horizon sensors come in scanning and non-scanning varieties that can have accuracies down to the tenth of a degree range [40], [50]. Merely detecting the presence of the Earth is insufficient [51], so they are typically designed to detect the Earth horizon and then determine a local vertical direction. Often the results from two sensors are averaged to minimize errors [27]. Interpretation of horizon sensor data can be complicated by the need to correct for Earth oblateness effects to achieve maximum accuracy [49]. Probably the most significant drawback for this application is the fact that their accuracy is limited by horizon definition and they degrade at lower altitudes, precisely the flight regime of sounding rockets.

Sun sensors are arguably the most common attitude sensor employed on both satellites and sounding rockets, at least for daytime launches [18], [52]. For night launches, a Moon sensor or horizon sensor may be substituted. The Sun is typically treated as a point source for low-altitude satellite and sounding rocket applications, subtending approximately .267 deg [52] or 7×10^{-5} steradians [51] at Earth orbit about the Sun. They come in both analog and digital designs. Expensive varieties can have accuracies as good as .01 deg, but this is difficult to achieve [49], [50]. Smith reports that a commonly used digital Sun sensor produces accuracies on the order of .5 degrees [39]. One significant advantage, for this effort, is that Sun sensors can be designed and built in-house, by university student programs, or other programs with smaller budgets. For example, the university designed and built Sun sensor flown on the Student Rocket Project 4 (SRP-4) mission achieved a reported accuracy of 4 degrees [53]. It is not clear how this reported "accuracy" is defined, but assuming "accuracy" can be interpreted as the 3σ value, meaning 99% of the values fall in this range, and assuming a Gaussian measurement error distribution, the standard deviation for this "accuracy" could be considered to be 1.33 degrees, or approximately 0.023 radians.

Star sensors are the most accurate attitude sensors available. While they are very accurate, they have the potential downside of being blinded by the Sun, Moon, or planets [49]. Star sensors are most effective when a fairly accurate estimate of attitude is already known, and the star sensor data is then used to refine the estimate [18]. Additionally, they are very complex and expensive, which eliminates them from practical consideration for this research effort.

Inertial Measurement Units, or IMUs, typically consist of a collection of gyroscopes and accelerometers and have the unique ability to determine position and attitude independent of outside sources of information. Once given an accurate initialization consisting of its position and attitude, and assuming perfect gyros and accelerometers, an IMU will accurately measure and report both position and attitude. This type of unit has been used successfully on many different platforms, the most notable being manned spacecraft, nuclear submarines, and intercontinental ballistic missiles. Unfortunately, components with sufficient accuracy to make IMUs practical are very expensive. Even given robust filtering and estimating techniques, and frequent reinitialization from outside reference sources, IMUs are impractical from a cost and manufacturing standpoint for a "low-cost" application. Recent developments in solid-state gyroscopes and accelerometers may make this a viable alternative in the future, but at this point in time, they are not a practical option for the work envisioned here.

While the quality of components needed to implement a practical IMU are cost prohibitive, gyroscopes can be used alone as attitude sensors. The two major categories are rate gyros that measure rotation rate, as their name implies, and rate integrating gyros that output an angular displacement from an initial value.

The less expensive versions of these gyroscopes have drift rates far in excess of what is acceptable for use as part of an IMU, but over short time steps, can provide adequate measurement of rotation rates to be used as inputs to a filtering algorithm or dynamics model. Rate integrating gyros are typically far more accurate than rate gyros, but are often far more expensive [54]. A problem with using gyros in any attitude determination system where Kalman-based filtering is used is that the low frequency noise component (also referred to as bias or drift) violates the white noise assumption required for Kalman filtering. Although this issue can sometimes be mitigated by forming a larger state vector augmented with the gyro noise, the result is a significantly more complex estimator [41]. Expensive gyros exhibit drift rates as low as .001 deg/hr, but recently developed solid state gyros, that would fit the budget of most "low-cost" programs, have drift rates as high as 300 deg/hr. While these high drift rates make their use questionable for use in a rate-integrating sense, three inexpensive rate gyros mounted orthogonally can provide noisy measurements of the body frame rates about each axis.

In addition to low cost gyros, an alternative low-cost source of rate measurements might be found in the recent development of inexpensive accelerometers. Again, these do not have sufficient accuracy for use in an IMU application, but they may provide useful inputs to a filtering algorithm such as that envisioned here. Rates could possibly be obtained either through the integration of the accelerometer outputs, or possibly by processing tangential and radial components of acceleration about each body axis.

Another commonly used sensor is the magnetometer. A three-axis magnetometer mounted at a known orientation to the vehicle body frame can measure the strength and direction of the local magnetic field vector. This allows for the determination of the spacecraft orientation relative to the magnetic field. Recent advances in modeling the Earth's magnetic field have improved the workability of magnetometers as attitude sensors. Reeves reports an accuracy of approximately 5 degrees at an altitude of 200 km [50]. Despite advances in field modeling, changes in the field due to solar effects and other perturbations limit the accuracy of these devices for measuring attitude. As with the Sun sensor, a significant advantage of this type of sensor for the work considered here is that it can be designed and manufactured in-house at reasonable cost by experimenters in a "low-cost" environment [53]. Additional benefits of magnetometers are that they are vector sensors, providing both the direction and magnitude of the magnetic field, reliability, light weight, low power requirements, wide temperature operating range, and they have no moving parts [55]. While they are sometimes limited in satellite applications by the $1/r^3$ dependence of field strength on distance from the source, for relatively low altitude sounding rocket missions, this is not an issue. Additional concerns include the effects of the magnetic environment within the vehicle and uncertainties in the magnetic field model. Chobotov claims that "magnetometers can provide directional accuracy to within one degree of the magnetic field vector [40]," and Fortescue and Stark report that accuracy is limited to approximately 0.5 degrees [25]. Herbert claims an accuracy of .25 degrees or better [4]. Low-cost versions of this instrument most likely will not achieve these accuracies. The three-axis magnetometer designed and built by the Tokai University and flown on Student Rocket Project 4 (SRP-4) demonstrated an accuracy of approximately 0.5 degrees [56]. As with the Sun sensor, in order to use a magnetometer to determine attitude, an accurate position of the vehicle is required in order to cross reference the measured field with that predicted by a model [57]. From a design and implementation standpoint, careful calibration must be accomplished taking into account magnetization due to the rocket itself and potential changes to the magnetic environment throughout the flight [3], [4].

Radio interferometer and GPS attitude determination are similar methods from a technology point of view. Each relies on the reception of a known radio signal at two antennas or at a single antenna at two distinct times. While these methods promise light weight and low cost, Eterno points out a number of challenges to successfully implementing them, among them are that "accuracy is limited when used as an attitude sensor by separation of antennas, the ability to resolve small phase differences, relatively long wavelength of signal and multipath issues [49]."

Yet another radio-based method makes use of directional antennas and signal strength to estimate attitude. The received signal strength of the vehicle telemetry signal at the ground station is used

> to determine vehicle aspect with respect to the vector describing the position of the vehicle as seen from the ground. Here the variation of signal strength as the vehicle precesses is used. This method has also had limited success since normally, the vehicle may precess only a few times in a flight and the signal strength variation due to precession is partially masked by the signal strength variation due to changes in vehicle range throughout the flight and other effects [5].

Under ideal conditions, this type of directional antenna method can produce accuracies of approximately .01-.5 deg (~1% of antenna beam width) [50].

In the end, what is required is a truly inertial system that can be initialized and then allowed to propagate the attitude without external inputs, or measurement of two or more external references at each time of interest from which an unambiguous estimate of the attitude at that moment can be computed [27], [49]. Based on the sensor characteristics outlined above, in conjunction with the goals of this research, the sensors best suited to this application are a Sun sensor, a three-axis magnetometer, and a set of three orthogonally mounted rate gyros.

6.3 Sensors Best Suited to This Problem: Sun Sensor, Magnetometer, Rate Gyro

Recognizing that a precise Inertial Measurement Unit is beyond the budget of many "low-cost" research efforts, a sensor suite capable of measuring direction to external references is dictated. A minimum of two nonparallel directions are needed to uniquely determine the rocket's orientation [27]. Culling the available sensor types and evaluating the current costs and state of technology results in a suite composed of a Sun sensor and a triaxial magnetometer to measure the two reference directions, and inexpensive rate gyros or accelerometers are a possibility to cost-effectively provide rotational rate information. While these are not strictly necessary for instantaneous attitude determination, many filtering algorithms use a dynamic model to refine the estimate of the attitude. As discussed in the section on dynamics, the angular rates determine how the vehicle behaves dynamically, so a measurement of these rates is useful. In fact, if perfect sensors provided the real values of the angular rates and an accurate initial position was available, "integrating the angular rates through Euler equations would lead us to determine the orientation at each time step [11]." While the rate sensors that can be afforded here will not be sufficient for such a process, they may provide adequate information to allow the filtering algorithm to further refine the estimates of attitude over what would be achieved using only the vector sensors.

The algorithm to be developed will focus on the manipulation of data from any two vector sensors. While the discussion above has narrowed the field to what would best suit this particular application, the algorithm could make use of others. For instance, in the case of a night launch, it might be necessary to substitute a star, Moon or a horizon sensor in place of the Sun sensor. Likewise, if some new rate sensor became available, it could be substituted for the rate gyro. In this sense, the algorithm is "blind" to where the data comes from, merely requiring the measured values and the quality of the measurements. One additional constraint on the choice of vector sensors is that a model must be available to provide values for the measured vector in the reference frame at the times of interest.

The development of the algorithm will proceed using a baseline of a Sun sensor with a measurement accuracy of 4 degrees, corresponding to a standard deviation of 1.333 degrees using the 3σ interpretation described above. This corresponds to what was reported by Tokai University based on their experience from the SRP-4 mission [53]. Based on the information in the previous section, performance at least this good should be expected. A triaxial magnetometer with a measurement accuracy of 10 degrees, corresponding to a standard deviation of 3.333 degrees using the 3σ approach, will be used as a baseline for the second vector sensor. This, or better, should be readily available from a magnetometer designed and built "in-house" at a university or similar program, considering the reported accuracy of the SRP-4 magnetometer at +/- 0.5 degrees in each axis. [56]. Given model uncertainty for the magnetic field and errors in the position solution of the rocket, a standard deviation of 10 degrees should be very conservative. Finally, algorithms will be designed that both use and do not use rate information such as that provided by the rate gyros, or accelerometers, described above. This will enable a comparison of what benefit, if any, is gained relative to the increased complexity of including the rate sensor in a sensor suite. As a baseline, a rate sensor with a measurement accuracy of 1 revolution per minute (rev/min), corresponding to a standard deviation of 0.333 rev/min will be used.

Due to the non-inertial nature of these sensors, an accurate estimate of the vehicle position and an accurate time reference are necessary for determination of rocket attitude. These are required to reference solar ephemeris and magnetic field models to determine the inertial reference vectors at the time of the estimate. This in turn enables the calculation of the pointing vector of the rocket, or the attitude. For this work, these position and time references are assumed to be available, either from GPS, RADAR, or other sources.

7.0 Applicable Sensor Fusion Principles

7.1 Introduction

In this chapter, I discuss several of the more prominent methods for "fusing" data that may be applicable to this problem. Herein I evaluate each and support the choice that will be used for this research effort. The term "sensor fusion" is somewhat ill-defined and can be presumed to include many things. In this work, I distinguish between "fusion" and "filtering," although some might include filtering as a type of fusion [58], [59]. I will discuss "filtering" methods as they apply to the rocket attitude determination problem in Chapter 8 of this work. In this chapter, I limit the discussion to those methods that operate on multiple noisy measurements taken simultaneously, as opposed to a method that somehow analyzes a time history of measurements. In the end, the algorithm developed for this effort will employ techniques from each group.

7.2 Sensor Fusion Techniques: Weighted Averaging, Error Checking, Error Minimization / Parameter Optimization

Among the data fusion techniques in common use, some are very simple while some are more sophisticated. "Techniques to combine or fuse data are drawn from a diverse set of more traditional disciplines, including digital signal processing, statistical estimation, control theory, artificial intelligence, and classic numerical methods [60]." The goal for this work, with respect to data fusion, is to survey the methods available and investigate what, if any, tools might be successfully applied to this effort.

Many of the fusion efforts described in the literature are designed to arrive at a better solution given multiple sensors measuring the same quantity [60], [61], [62]. For example, if you have two or more Sun sensors and you combine them in an intelligent way, you may be able to get a better answer than you would from a single sensor, owing to the additional information available. One common approach to this is a type of weighted averaging summarized by Perrella [61] wherein the overall estimate is achieved by adding the multiple measurements together, after suitably weighting them based on some estimate of their relative quality. Typically, the "quality" of an estimate is measured by its standard deviation or variance. Obviously, it is desirable that the standard deviation of the fused estimate be less than that of the individual estimates. Perrella points out that a careful choice of weights often achieves this, while a poor choice might actually do more harm than good. As an example of this type of fusion, in an extension of the two sensor case described by Perrella, the "fused" estimate from three sensors is calculated as

$$X_{fused} = W_1 * X_1 + W_2 * X_2 + (1 - W_1 - W_2) * X_3, \tag{7.1}$$

yielding a variance of the fused estimate determined by

$$\sigma_{fused}^2 = (W_1\sigma_1)^2 + (W_2\sigma_2)^2 + [(1 - W_1 - W_2)\sigma_3]^2 . \tag{7.2}$$

Clearly, the goal is to choose the weights so as to minimize the variance of the fused estimate. Taking the gradients of (7.2) with respect to the weights, setting them equal to zero, and solving yields the following expressions to calculate the weights W_1, W_2 and W_3 to minimize σ^2_{fused}.

$$W_1 = \frac{\sigma_2^2}{\sigma_1^2 + \sigma_2^2 + \frac{\sigma_1^2\sigma_2^2}{\sigma_3^2}} \tag{7.3}$$

$$W_2 = \frac{\sigma_1^2}{\sigma_1^2 + \sigma_2^2 + \frac{\sigma_1^2\sigma_2^2}{\sigma_3^2}} \tag{7.4}$$

$$W_3 = 1 - W_1 - W_2 . \tag{7.5}$$

While such an approach has potential benefits in an application similar to that devised in this effort, in order to minimize complexity and cost, only one sensor is employed to measure each quantity here. This and other methods for intelligent synthesis of multiple sensors warrant further investigation for an application where the budget allows such a configuration. It has been shown that, "a statistical advantage is gained by adding the *N* independent observations ... assuming the data are combined in an optimal manner [60]." Fortunately, he also points out that a statistical advantage may also be obtained by intelligently combining *N* observations, over time, from an individual sensor. Rather than employ redundant sensors, at additional cost and system complexity, it is this second approach, in the form of "filtering" that is implemented in this work. The details are described in the next chapter on filtering.

In addition to the risk of doing more harm than good through a poor choice of W in the data fusion method above, there are other things that cause data fusion approaches to go awry. Hall and Garza have enumerated the most common pitfalls and the following from their list are the most relevant to this task:

> - There *is no substitute for a good sensor.* No amount of data fusion can substitute for a single, accurate sensor that measures the phenomena that you want to observe.
> - *Downstream processing cannot make up for errors (or failures) in upstream processing.* Data fusion processing cannot correct for errors in processing (or lack of preprocessing) of individual sensor data.

> - *Sensor fusion can result in poor performance if incorrect information about sensor performance is used.* A common failure in data fusion is to characterize the sensor performance in an ad hoc or convenient way. Failure to accurately model sensor performance will result in corruption of the fused results.'
> - *There is no such thing as a magic or golden data fusion algorithm.* Despite claims to the contrary, there is no perfect algorithm that is optimal under all conditions. Often, real applications do not meet the underlying assumptions required by data fusion algorithms (e.g., available prior probabilities or statistically independent sources).
> - *Quantifying the value of a data fusion system is difficult.* A challenge in data fusion systems is to quantify the utility of the system at a mission level. Although measures of performance can be obtained for sensors or processing algorithms, measures of mission effectiveness are difficult to define [63].

For this effort, these translate into an evaluation of how well the algorithm performs given a particular sensor quality, a necessity to handle bad or missing data as it is presented to the algorithm, evaluation of the impact of inaccuracies in the assessment of the sensor quality, and the investigation of more than one fusion/filtering method in order to gauge the effectiveness of the final choice.

Another form of processing that is not often associated with data fusion is error minimization. For this problem, however, an error function is defined, to be detailed later, that incorporates two sensor measurements. Error minimization is used to combine these sensor measurements into a reduced "measurement" that minimizes the defined error function. In this respect, error minimization becomes "data fusion." In cases such as this where an error function can be devised, numerous optimization techniques have been developed to minimize the error function. One method that has found widespread use in the field of digital signal processing is the "approximate steepest descent" method known as Least Mean Square (LMS). Introduced by Bernard Widrow and Marcian Hoff in 1960, the LMS algorithm seeks to minimize the performance index of mean square error [64], [65]. Other commonly used methods for minimizing performance indices are true "steepest descent" methods, "conjugate gradient" methods and variations of "Newton's" method. As pointed out by Hagan et al. [64], steepest descent is the simplest and is almost certain to converge, but is often slow. Newton's method is much faster, but requires the calculation of matrix inverses and Hessian matrices. Conjugate gradient approaches are a compromise that do not require second derivatives, but still converge to the minimum of a quadratic performance index in a finite number of steps [64]. Here I have listed only a sample of all the techniques available, and it should be clear that there are many suitable approaches for finding the minimum of a quadratic error surface such as mean-square-error (MSE). It will be shown that the performance surface of interest here is indeed a "squared-error" surface. The application investigated in this thesis, however, requires not only accurate minimization of this quadratic error surface, but also speed. The issue of speed is driven by the desire to develop an algorithm sufficiently fast to be run in real time, with minimum computer resources.

This requirement for a fast routine narrows the field of suitable methods somewhat. While there are still numerous choices that could be argued to be superior, the variations on the Newton method have the desired characteristics of being relatively fast, having the quadratic convergence property, and are not overly complex. As derived in detail by Hagan et al. [64], the Newton method locates a stationary point for quadratic functions in one iteration, but has the disadvantage of requiring the computation and storage of the Hessian matrix. Since this matrix is composed of second-order derivatives, its calculation can be a difficult and time-consuming process if the functions are complicated or not well known. It is also pointed out that the convergence properties of Newton's method can be "quite complex." The Gauss-Newton method is a modification that overcomes some of the disadvantages of the standard method. The main advantage of Gauss-Newton over the standard Newton's method is that it still yields rapid convergence, but does not require calculation of second derivatives. This is due to the fact that only the Jacobian must be calculated, not the Hessian. Based on its characteristics, I have concluded, as did the authors in [12], that Gauss-Newton is best suited to the application in this thesis.

The Gauss-Newton method, as developed by [64], begins with the standard Newton method. While its exact form is detailed in a later section, the performance index, *F*(*q*), is a sum of squares that is a function of the attitude quaternion, *q*. The quaternion that optimizes the performance index is found iteratively using

$$q_{k+1} = q_k - A_k^{-1} g_k \tag{7.6}$$

where $A_k = \nabla^2 F(q)\big|_{q=q_k}$ and $g_k = \nabla F(q)\big|_{q=q_k}$. As is the case for this application, when the performance index is a sum of squares, *F(q)* may be represented as a function of the error vector, ε, as

$$F(q) = \sum_{i=1}^{N} \varepsilon_i^2(q) = \varepsilon^T(q)\varepsilon(q), \tag{7.7}$$

and the *j*th element of the gradient is

$$[\nabla F(q)]_j = \frac{\partial F(q)}{\partial q_j} = 2\sum_{i=1}^{j} \varepsilon_i(q)\frac{\partial \varepsilon_i(q)}{\partial q_j}. \tag{7.8}$$

This means that the gradient can be written in matrix form as

$$\nabla F(q) = 2J^T(q)\varepsilon(q) \tag{7.9}$$

where

$$J(q) = \begin{bmatrix} \frac{\partial \varepsilon_1(q)}{\partial q_1} & \frac{\partial \varepsilon_1(q)}{\partial q_2} & \cdots & \frac{\partial \varepsilon_1(q)}{\partial q_n} \\ \frac{\partial \varepsilon_2(q)}{\partial q_1} & \frac{\partial \varepsilon_2(q)}{\partial q_2} & \cdots & \frac{\partial \varepsilon_2(q)}{\partial q_n} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial \varepsilon_N(q)}{\partial q_1} & \frac{\partial \varepsilon_N(q)}{\partial q_2} & \cdots & \frac{\partial \varepsilon_N(q)}{\partial q_n} \end{bmatrix} \tag{7.10}$$

is the Jacobian matrix. For the standard Newton's method, you next must calculate the Hessian matrix:

$$[\nabla^2 F(q)]_{k,j} = \frac{\partial^2 F(q)}{\partial q_k \partial q_j} = 2\sum_{i=1}^{N}\left\{\frac{\partial \varepsilon_i(q)}{\partial q_k}\frac{\partial \varepsilon_i(q)}{\partial q_j} + \varepsilon_i \frac{\partial^2 \varepsilon_i(q)}{\partial q_k \partial q_j}\right\}. \tag{7.11}$$

This can be written in matrix form as

$$\nabla^2 F(q) = 2J^T(q)J(q) + 2S(q), \tag{7.12}$$

where

$$S(q) = \sum_{i=1}^{N}\varepsilon_i(q)\nabla^2 \varepsilon_i(q). \tag{7.13}$$

If S(*q*) is assumed small, the Hessian is approximated as

$$\nabla^2 F(q) \cong 2J^T(q)J(q) \tag{7.14}$$

substituting equations (7.14) and (7.9) into equation (7.6) yields the Gauss-Newton method:

$$\begin{aligned} q_{k+1} &= q_k - [2J^T(q_k)J(q_k)]^{-1} 2J^T(q_k)\varepsilon(q_k) \\ &= q_k - [J^T(q_k)J(q_k)]^{-1} J^T(q_k)\varepsilon(q_k). \end{aligned} \tag{7.15}$$

Here it is clear that Gauss-Newton only requires first-order derivatives in the Jacobian, as opposed to the second-order derivatives necessary to calculate the Hessian for the standard Newton method. Two disadvantages of this algorithm are that the matrix $H=J^TJ$ may not be invertible, and that storage space must be allocated for this matrix. If inversion of the *H* matrix becomes problematic, a modified Gauss-Newton method called the Levenberg-Marquardt algorithm may be used to guarantee invertibility [64].

7.3 Wahba's Problem and Gauss-Newton Iteration

The approach in this thesis for determining the attitude quaternion follows the problem first published in 1965 by Grace Wahba [66]. In this problem, "Wahba proposed an attitude solution by matching two non-zero, non-collinear vectors that are known in one coordinate frame, and measured in another [14]." The details of the development follow in a later chapter, but it will be seen that an error function is devised that is based on the differences between known and measured vectors. This error function is of the form [11]

$$F(q) = \bar{\varepsilon}^{T}\bar{\varepsilon} = (\bar{\mathrm{y}}^{I} - \mathrm{M}(q)\bar{\mathrm{y}}^{B})^{T}(\bar{\mathrm{y}}^{I} - \mathrm{M}(q)\bar{\mathrm{y}}^{B}) \tag{7.16}$$

where

$F(q)$ = error function to be minimized,

$\bar{\varepsilon}$ = error vector (6×1),

$\bar{\mathrm{y}}^{I}$ = reference vector in the "inertial" frame (6×1),

$\mathrm{M}(q)$ = transformation matrix from "body" to "inertial" frame (6×6),

$\bar{\mathrm{y}}^{B}$ = measurement vector in the "body" frame (6×1).

The reference vector in this case is a concatenation of the Sun vector in the inertial frame and the magnetic field vector in the inertial frame. These vectors come from models or look-up tables at each time step. The measurement vector is a concatenation of the Sun and magnetic field vectors measured in the body frame by the sensors. The "front end" of the algorithm will seek to minimize this error function. Examination of *F(q)* reveals that it is a quadratic function, and certain methods are geared toward, or better suited for, the manipulation of quadratics. This being the case, a driving factor in selecting a data fusion approach is to choose one that efficiently and successfully accomplishes this minimization.

As discussed in the previous section, the methods best suited to minimizing a quadratic function, as presented here, are the LMS method and the variations on Newton's method. Based on the need for rapid convergence and a minimized computation, in order to facilitate real-time implementation, the Gauss-Newton approach stands out at the most promising method and will be implemented in this algorithm. A detailed development of this implementation and an in-depth investigation of its convergence properties for this application will be covered in later sections of this work.

Having selected a promising sensor fusion method from the many available, it is now necessary to evaluate "filtering" methods that use information from data taken over time in order to provide a best estimate at a given time of interest. The next chapter steps through the evaluation of a number of those methods to determine the best approach for this effort.

8.0 Applicable Filtering Techniques

8.1 Introduction

As outlined in the last chapter, a distinction is made between data fusion and filtering in this work. Filtering describes the combination of measurements gathered over time to generate the best estimate of a particular quantity of interest. These measurements may come from a single sensor or from multiple sensors. The last chapter investigated those data fusion methods that lend themselves to the solution of the problem under study. This chapter takes a similar survey of filtering techniques that are most applicable to the low-cost rocket attitude determination problem.

8.2 Promising Filtering Techniques

There are many filtering approaches, and variations on well-known techniques, that have been devised to handle the state estimation problem [67]. The goal of this research effort, with respect to filtering, is to evaluate the most promising techniques and to identify and implement one or more that provide the most leverage with respect to yielding the best attitude estimate given the constraints of the low-cost approach. In addition to being constrained to relatively noisy sensors and a limited number of them, there are the challenges of the problem itself. Sounding rockets are highly dynamic vehicles and their attitude state can change rapidly and over a broad range of values. Additionally, the equations governing the solution for attitude are nonlinear by nature. These problem parameters limit the number of filtering approaches that will enjoy success for this application. While many approaches were reviewed, this section discusses 7those that held out the most promise and evaluates their suitability for this problem.

8.2.1 Extended Kalman Filter

Despite the recent appearance of alternatives for the filtering of nonlinear systems, the extended Kalman filter (EKF) remains the most popular approach due to its simplicity, familiarity, and successful application [68], [69], [70]. Entire works have been devoted to the roots of, modifications to, and characteristics of the Kalman filter. Three excellent references are [17], [71] and [72], and there are many more. Here I will focus on a brief history and a summarization of the characteristics of the Kalman filter as they apply to this problem.

The original formulation of the Kalman filter is attributed to R. E. Kalman and was first published in 1960 [73]. In its original form, it was designed as a statistical approach to estimate the state of a system described by linear differential equations, expressed in state-space form, where the measurements are linear functions of the state [17], [74]. Such a system may be expressed as:

$$\begin{aligned} \dot{\bar{x}} &= F\bar{x} + \bar{w} \\ \bar{z} &= H\bar{x} + \bar{v} \end{aligned} \tag{8.1}$$

where $\bar{x}$ is the system state vector, F is the system fundamental or dynamics matrix, $\bar{w}$ is a process noise vector, H is the measurement matrix, $\bar{z}$ is the measurement vector, and $\bar{v}$ is a vector representing the measurement noise components.

One of the great advantages, for the linear system case, is that the prediction step makes use of the state transition matrix to propagate the system states forward using only matrix multiplication [17]. In estimating the state vector at the next time step, a Kalman filter "utilizes a weighting function, called the Kalman gain, which is optimized to produce a minimum error variance. For this reason, the Kalman filter is called an optimal filter. For linear system models, the Kalman filter is structured to produce an unbiased estimate [74]."

While this is a very powerful filtering method, its traditional form is limited to linear problems, or those that are only slightly nonlinear. Since many real-world problems, including attitude determination of a sounding rocket, are characterized by either nonlinear system equations, or nonlinear measurement equations, a method suitable to handling these problems was needed.

Very early on, the standard Kalman filter structure was modified to handle nonlinear problems. Two approaches are commonly taken. The first is known as a linearized Kalman filter in which the system model is linearized about a nominal trajectory before filtering begins. This approach is often robust with respect to poor initialization, but does not respond well if the actual trajectory being estimated does not closely match that used to linearize the system up front [17]. Since it is feasible that a rocket's attitude will deviate from the expected motion, the linearized Kalman filter is not well-suited to this problem. A second, more common, approach is known as the extended Kalman filter. This development is attributed to Schmidt and is sometimes called the Kalman-Schmidt filter [71]. In this case, the trajectory is linearized about the estimated trajectory, at each time step, and evaluated using the current estimate of the state vector. While this approach is more susceptible to initialization problems, it is more robust in situations where the actual trajectory varies significantly from the expected trajectory [17], [71]. This is the approach that will be evaluated in this work.

The extended Kalman filter operates on a representation for a nonlinear system that is similar to the representation for a linear system found in (8.1)

$$\dot{\bar{x}} = f(\bar{x}) + \bar{w}$$
$$\bar{z} = h(\bar{x}) + \bar{v} \tag{8.2}$$

where $f(\bar{x})$ and $h(\bar{x})$ may now be nonlinear functions of the state vector. The extended Kalman filter is a recursive method of solving the nonlinear system indicated above. It is recursive in the sense that

> Each updated estimate of the state is computed from the previous estimate and the new input data, so only the previous estimate requires storage. In addition to eliminating the need for storing the entire past observed data, the Kalman filter is computationally more efficient than computing the estimate directly from the entire past observed data at each time step of the filtering process [67].

While a more detailed development will follow in Chapter 9, a brief overview of the filter structure is given here. Following the development in [17], the discrete extended Kalman filter can be considered to be based on the following recursive equations. The new estimate of the state vector, $\hat{x}$, is formed at each time step by updating the projected value of the state vector, $\bar{x}$, with a correction based on the projected observations, $\bar{z}$, relative to the actual observations, z^{*}, according to

$$\hat{x} = \bar{x} + K(z^{*} - \bar{z}) \tag{8.3}$$

where *K* is known as the Kalman gain. The recursive equations serve the purpose of generating each of the terms in (8.3) in order to calculate the new estimate, $\hat{x}$. There are three main equations, known as the Riccati equations that are calculated during each iteration, or time step. The first calculates what is known as the covariance of the error in the estimate of the state vector, *M.* This is further qualified to define *M* as the covariance of the error in the estimate *before* the update, meaning during the current time step, but before the new observation has been processed by the filter. Essentially, *M* is the filter's own assessment of how good its estimate is. It is calculated as

$$M_k = \Phi_k P_{k-1} \Phi_k^T + Q_k \tag{8.4}$$

where Φ_k is the state transition matrix at time step *k*, P_{k-1} is the covariance of the error in the estimate at the previous time step *after* the update, or when the observation had already been processed by the filter, and Q_k is the discrete process noise matrix, which is yet another covariance matrix that represents the filter's assessment of how good the embedded dynamic model, or process is. As noted earlier, in a linear system, the state transition matrix may be used to calculate the state vector at any other point in time, through matrix multiplication. Unfortunately, for the EKF, this is no longer possible, because the state transition matrix is based on a truncated Taylor series expansion that results in a first-order linearization of the actual nonlinear system dynamics equations represented by $f(\bar{x})$. From linear systems theory, the continuous time state transition matrix may be represented as

$$\Phi(t) = e^{Ft} \tag{8.5}$$

where F is the linearized dynamics matrix for the system found using

$$F = \left.\frac{\partial f(x)}{\partial x}\right|_{x=\hat{x}} . \tag{8.6}$$

This equation is evaluated at the state estimate at each time step to get a new linearization of the dynamics matrix. Equation (8.5) represents the continuous time version of the state transition matrix, and the matrix exponential may be represented as an infinite Taylor series expansion of the form

$$\Phi(t) = e^{Ft} \approx I + Ft \tag{8.7}$$

where only the first two terms of the expansion are retained. This amounts to an additional linearization that assumes that the filter time step is sufficiently small such that the linearization is valid. Through extensive research, Zarchan has demonstrated that for most cases, no significant performance gain is achieved by retaining additional terms [17]. This may be traced back to the fact that the state transition matrix is only used as a means to calculate the Kalman gain, and not to propagate the states forward as would be the case in a linear system. For the EKF, the states are propagated through the numerical integration of the actual nonlinear dynamics equations embodied in $f(\bar{x})$ because this is more accurate than propagating through matrix multiplication using a linearized state transition matrix. The discrete process noise matrix, Q_k, is a function of the continuous time state transition matrix, $\Phi(t)$, and the continuous time process noise matrix, $Q(t)$, which is yet another covariance matrix defined as

$$Q(t) = E[ww^T]. \tag{8.8}$$

It is intended to represent the uncertainty in the embedded process model, or how good the dynamics equations are. The initialization of this and the other matrices in this section is discussed in Chapter 9. The discrete form of the process noise matrix is recalculated at each iteration to update the filter's assessment of how well the embedded model is performing. This done using the equation

$$Q_k = \int_0^{T_S} \Phi(\tau)Q(\tau)\Phi^T(\tau)dt . \tag{8.9}$$

The remaining elements needed to calculate the covariance before the update, M, is the covariance after the update from the last time step, P, and the discrete form of the state transition matrix, Φ_s. Φ_s is found by merely substituting T_s, the filter time step, in for continuous t in (8.7). P is the filter's assessment of how good its estimate of the state vector is after it has processed the new observation. This covariance matrix, which may be interpreted as a diagonal matrix with the squared error of each state estimate along its diagonal, is calculated at each iteration using

$$P_k = (I - K_k H) M_k \tag{8.10}$$

where I is the identity matrix and H is the linearized measurement sensitivity matrix found by linearizing *h(x)* from (8.2) by taking partials similar to what was done to get F in (8.6). This is the second of the Riccati equations that are iterated in a traditional Kalman filter. The last of the three is the equation that calculates the Kalman gain at each time step. It is found using

$$K = MH^T [HMH^T + R]^{-1} \tag{8.11}$$

where all but R have already been defined. R is the last of the covariance matrices and represents the uncertainty, or error, in the measurements. It is defined as

$$R = E[vv^T] \tag{8.12}$$

where v is the vector from (8.2) that represents the statistical noise process that corrupts the measurements. Given the expectation operator in (8.12), and assuming a zero mean sensor error, the diagonal elements of this matrix may be interpreted as the variances of the sensor measurements. Or similarly, the square root of the diagonal elements may be interpreted as the standard deviations of the sensor measurements. In fact, this is how this particular matrix is initialized as will be detailed in the next chapter.

Looking back to (8.3), we see that the whole point of this endeavor is to generate a new estimate of the state vector, $\hat{x}$. $\bar{x}$ is the projected state vector at the next time step and is found by numerically integrating the nonlinear state equations forward one time step. $\bar{z}$ is the projected observation and is found, in general, by numerically integrating the nonlinear measurement equations from (8.2) forward one time step, and evaluating the result at the projected values of the states. Finally, z^* is the actual observation, usually coming from the sensors. With all of these pieces in place, a new state estimate is formed, and the process begins again.

The Kalman filter is often proclaimed to be an "optimal" filtering method for state estimation and in fact it is the "minimum mean-square (variance) estimator of the state of a linear dynamical system [67]." As Gelb points out, however, "The truly *optimal* filter must model *all* error sources in the system at hand...Also, it is assumed in the filter equations that exact descriptions of system dynamics, error statistics and the measurement process are known [75]." These two assumptions must certainly be violated in every real world problem. A key example is the treatment of both the process noise, $\overline{w}_r$, and the measurement noise, $\overline{v}_r$. These are modeled as white, Gaussian and zero-mean and two fundamental assumptions of the Kalman filter are that the unmodeled measurement errors are white (i.e., uncorrelated), and the unmodeled error dynamics are white (i.e., uncorrelated) [76]. While these may be good assumptions from the perspective that using them produces acceptable results, they are most likely not true in a strict sense.

It is clear from this development that the EKF is not really a "nonlinear" filtering method, but really a modification of a linear approach to handle a nonlinear problem. Unfortunately, despite being the most common, and perhaps best known, filtering technique for nonlinear problems, the EKF has a number of well established shortcomings [69] First, the linearization required to apply the Kalman filter structure developed for linear systems to nonlinear systems can result in highly unstable filters. As Julier and Uhlmann point out, this is especially true if the assumption of local linearity is violated. Risk of this is minimized if very short time steps, relative to the time scales associated with changes in the states, are used. Second, "linearization can be applied only if the Jacobian matrix exists, and the Jacobian matrix exists only if the system is differentiable at the estimate [69]." Additionally, calculation of the Jacobian, if it does exist, is often not a trivial undertaking. At the very least, it requires a significant amount of algebra as evidenced by the derivation for this application, to be seen in Chapter 9. Haykin highlights another well-known liability of the EKF, that it is prone to stability problems due to numerical difficulties, with the following:

> For example, the posteriori covariance matrix $\mathbf{P}_k$ is defined as the difference between two matrices...Hence, unless the numerical accuracy of the algorithm is high enough, the matrix $\mathbf{P}_k$ resulting from this computation may *not* be nonnegative-definite. Such a situation is clearly unacceptable, because $\mathbf{P}_k$ represents a covariance matrix. The unstable behavior of the Kalman filter, which results from numerical inaccuracies due to the use of finite-word length arithmetic, is called the *divergence phenomenon* [67].

Numerical issues are also noted in [17], [71], and [77]. Additional implementation concerns, timing issues, and tuning problems appear in the literature as well [69]. All of these issues can contribute to decreased accuracy of the state estimate produced by the EKF. Despite its successful application in many nonlinear applications, the drawbacks noted here give impetus to the search for nonlinear filtering methods that are not a "linear method made to work."

8.2.2 Variations on the Extended Kalman Filter: Adaptive Kalman Filter / Multiple-Model Filter

A variation on the EKF is the Adaptive Kalman Filter. While the overall structure is the same as that described in the preceding chapter, a Kalman filter may be made "adaptive" by including some of the dynamic equation parameters, or noise parameters, in the state vector [78], [79], [80]. By increasing the size of the state vector in this fashion, one or more of the parameters embedded in the state dynamic equations are allowed to change and their value is estimated at each filtering time step. Then the new parameter values are used for the next iteration.

This approach is probably most useful in situations where the dynamic model is expected to change significantly over time. For example, if a vehicle's structure changes by losing a nose-cone, one would expect that the coefficient of drag for the vehicle will change significantly. If the vehicle's position and velocity states are being estimated, the filter will likely rely on a kinematic equation of motion to predict the position and velocity at the next time step [17]. This dynamic model almost certainly will incorporate the coefficient of drag. Including the coefficient of drag as an estimated state might help the filter better estimate the position and velocity states. Of course, increasing the size of the state vector also increases the dimension of many of the matrices used for a Kalman style filter, thereby increasing the computational load. This approach is best suited to situations where the model is well understood, as in the example above, but where parameter values may change. As discussed in Chapter 5, the dynamic model for the rocket's rotational motion is complex, and the desire is to make this algorithm both time efficient for real time implementation and easily transportable between different vehicles. As a result, a generic, simplified dynamic model is employed, and the approach described here is not well-suited to this effort.

Another variation on an "adaptive"implementation of a Kalman style filter is the Multiple Model Adaptive Estimator (MMAE) approach described by Hanlon and Maybeck [81]. Similar to the adaptive Kalman filter described above, this approach handles the situation where the general model is known, but the exact model is not known. Instead of attempting to estimate the unknown parameters in the model, multiple Kalman style filters are implemented in parallel, each with a different model. Then a "hypothesis" test is accomplished on the results from each filter in order to determine which provides the best results. The drawback for this application is the additional computing power needed to implement multiple filters in parallel. It also requires higher fidelity models than the model to be implemented here. This approach is applied in [81] to flight control sensor/actuator failure detection and identification and a similar approach is described by Rama for application to target tracking [82]. Multiple model filtering seems best suited to such applications where several realistic models can be produced to cover different situations. Additionally, each individual filter suffers from the same drawbacks described for the individual filter in the previous section. For these reasons, this approach is not pursued for this work.

8.2.3 Alternatives to the Extended Kalman Filter

A number of other filtering approaches were investigated for this nonlinear problem. Having established that the EKF is the traditional approach, the goal was to identify any other promising technique that could be applied here to better estimate rocket attitude given the problem constraints.

One alternative to the EKF is the complementary filter. This approach has been used successfully in lieu of an EKF by several authors. One example is described in [15] where a complementary filter successfully combines data from accelerometers, magnetometers and rate sensors to generate an attitude estimate. The rate sensor data is used to determine orientation of the object, but due to sensor accuracy, this is only possible for short periods. This problem is overcome by using "complementary" data from other sensors to "recalibrate" the solution and combining both sets of data using the complementary filter structure [15]. This type of filter is typically used when statistical properties of the measurements and dynamic processes are not known to sufficient accuracy [13]. McGhee et al. [13], also point out that the most serious drawback to "optimal" filters such as the EKF is that they are highly "tuned" to the assumed problem statistics. The complementary filter structure simplifies the "tuning" process by employing a single parameter which is adjusted for best performance, as opposed to several parameters in the case of the EKF. For this application, the statistics are expected to be fairly well defined, and it is desirable to estimate not only orientation, but also attitude rates. As opposed to correcting the solution from one set of sensors periodically, a solution as close to "optimal" is desired. Given this scenario, and lacking any noteworthy performance increase over the EKF, the traditional EKF appears to be better suited to this problem.

Another area that was reviewed which has received a great deal of recent interest, and shows much promise for certain applications, is "fuzzy logic." The proclaimed advantages over traditional methods include applications where the system is difficult to model, or where ambiguity or vagueness is common. Fuzzy logic can be described as a structured, model-free estimator that approximates a function through linguistic input/output associations. In this sense, fuzzy systems are rules-based as opposed to more traditional methods that use strict decisions and assignments [83]. For example, a room might be described as "hot" instead of as having a temperature > 100 degrees. Decisions for control or estimation would then be based on this linguistic interpretation of the situation. Given the desire to devise a transportable algorithm to be used for a wide variety of rockets, this model-free approach seemed promising, at first. Upon further investigation, however, it is unclear how such a linguistic interpretation might be applied to this problem and other, more traditional, approaches appear to be more promising.

Finally, relatively recent, and potentially revolutionary, developments in nonlinear filtering are the so-called "derivative-free" filtering methods. "Derivative-free" refers to the fact that no explicit Jacobian or Hessian matrices must be calculated as is the case when linearizing to implement an EKF, for instance.

The most notable among these is the Unscented Kalman Filter (UKF). Julier and Uhlmann describe the unscented transform (UT), upon which the UKF is based, as

> a mechanism for propagating mean and covariance information through nonlinear transformations…(that) is more accurate, is easier to implement, and uses the same order of calculations as the EKF. Furthermore, the UT permits the use of Kalman-type filters in applications where, traditionally, their use was not possible [69].

The UT is a new method for handling the nonlinear transformation, superior to linearization in many respects, that is "founded on the intuition that *it is easier to approximate a probability distribution than it is to approximate an arbitrary nonlinear function or transformation.*[69]" In their analysis of this recently developed transformation, Julier and Uhlmann go on to detail a number of the characteristics that make the UKF superior to the EKF for problems that embody a nonlinear transformation:

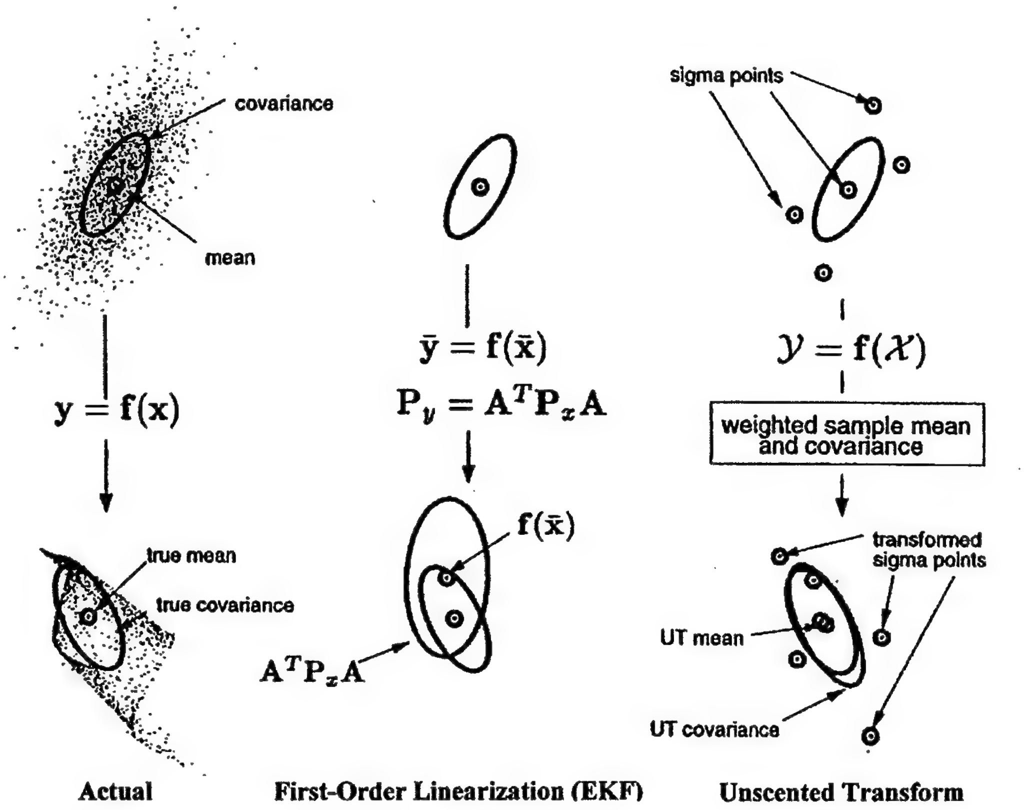
> - The UT is demonstrably superior to linearization in terms of expected error for all absolutely continuous nonlinear transformations. The UT can be applied with nondifferentiable functions in which linearization is not possible.
> - The UT avoids the derivation of Jacobian (and Hessian) matrices for linearizing nonlinear kinematic and observation models. This makes it conducive to the creation of efficient, general-purpose "black box" code libraries.
> - Empirical results for several nonlinear transformations that typify those arising in practical applications clearly demonstrate that linearization yields very poor approximations compared to those of the UT [69].

While Julier and Uhlmann are credited with the original development of the UKF, Wan and van der Merwe are recognized as having further developed the concept [68]. Wan and van der Merwe give an excellent synopsis of the differences between the UKF and the EKF:

> The basic difference between the EKF and UKF stems from the manner in which Gaussian random variables (GRV) are represented for propagating through system dynamics. In the EKF, the state distribution is approximated by a GRV, which is then propagated analytically through the first-order linearization of the nonlinear system. This can introduce large errors in the true posterior mean and covariance of the transformed GRV, which may lead to suboptimal performance and sometimes divergence of the filter. The UKF addresses this problem by using a deterministic sampling approach. The state distribution is again approximated by a GRV, but is now represented using a minimal set of carefully chosen sample points. These sample points completely capture the true mean and covariance of the GRV, and, when propagated through the *true* nonlinear system, capture the posterior mean and covariance accurately to second order (Taylor series expansion) for *any* nonlinearity. The EKF, in contrast, only achieves first-order accuracy.

> No explicit Jacobian or Hessian calculations are necessary for the UKF. Remarkably, the computational complexity of the UKF is the same order as that of the EKF [68].

Essentially, this new approach to propagating a mean and covariance through a nonlinear transformation hinges on choosing a set of sample points that accurately captures the information, propagating the points through the actual nonlinear transformation, and finally calculating weighted summations of the propagated points to estimate the new mean and covariance at the next time step. This process is illustrated for a two dimensional case by Figure 8.1 below, taken from [68]. This figure compares the


covariance
mean
$y = f(x)$
true mean
true covariance
$\bar{y} = f(\bar{x})$
$P_y = A^T P_x A$
$f(\bar{x})$
$A^T P_x A$
sigma points
$\mathcal{Y} = f(\mathcal{X})$
weighted sample mean and covariance
transformed sigma points
UT mean
UT covariance

Actual **First-Order Linearization (EKF)** **Unscented Transform**

Figure 8.1: Example of the UT for Mean and Covariance Propagation (Figure 7.3 in [68])

mean and covariance propagation performance for the EKF and the UT relative to the actual which was generated using a Monte Carlo approach. The UT may be implemented as part of a recursive algorithm, using a structure similar to that of the EKF, to create what is known as the Unscented Kalman Filter, or UKF. In this fashion, the advantages of the Kalman style filter are retained, while the method of propagating the mean and covariance are improved.

While the UKF appears to have numerous well-documented performance advantages over the EKF [68], [69], [84], Wan and van der Merwe point out a number of limitations. One significant point is that the UKF, like the EKF, still assumes a Gaussian distribution for the probability density function of the state variables. This limitation is mitigated by the fact that, as has been demonstrated with the EKF [17], this assumption often proves to be an adequate reflection of reality and "numerous real-world applications have been successfully implemented based on this assumption [68]." It is in the treatment of, or more accurately the propagation of, these random variables that the UKF excels in comparison to the EKF. The UKF calculates the mean and covariance "to second order or better which means more accurate implementation of the optimal recursive estimation equations, which is the basis for both the EKF and UKF [68]." While the UKF equations appear to be more complicated, both [68] and [69] point out that the number of calculations for both the EKF and the UKF is on the order of L^3, where L is the dimension of the state vector, when implemented as a state estimator. The apparent complexity of the equations is offset by the elimination of many of the matrix derivations and partial derivatives necessary when implementing the EKF. The UKF does incur a computation penalty, with respect to the EKF, in terms of propagating all the sigma points through the nonlinear transformation. This is computationally intensive since each is instantiated through the nonlinear dynamic equations using numerical integration. Finally, the UKF has three parameters that must be defined that govern how the sample set of points is chosen and propagated forward. The process for determining these parameters is not yet well delineated, and defining these parameters leads to some additional uncertainty in the implementation of the UKF [68]. More information on this "tuning" aspect of the UKF is found in Chapter 10.

8.3 Most Suitable Filtering Technique for this Application: Unscented Kalman Filter

Given the discussion of Section 8.2, the Extended Kalman Filter (EKF) and the Unscented Kalman Filter (UKF) were selected as the most suitable approaches for the low-cost attitude determination problem. While the benefits described for the UKF make it appear to be the obvious choice, the goal of this work is to determine the method that generates the best answer given the constraints of the low-cost system. With this goal in mind, both EKF-based algorithms and UKF-based algorithms are developed. In each case, versions that accept rate measurements from rate sensors and versions that rely on rate "measurements" from differencing quaternion elements and then converting to body rates are designed. This provides an opportunity to evaluate the performance of the UKF-based algorithm against what has traditionally been the standard approach. Furthermore, this allows each algorithm to process the exact same data under controlled circumstances for a true evaluation of not only which algorithm performs the best, but what the absolute quality of the performance is. Chapter 9 describes the detailed design of each of these algorithms, with additional information on "tuning" of the filters found in Chapter 10.

9.0 Devising the Algorithm

This chapter builds on the background information laid out in Chapters 3 through 8. Whereas those chapters summarized the survey of current and promising approaches, this chapter details the approach as implemented in design of the algorithms in this work.

9.1 Restating the Objective

It is important to maintain a clear view of the objective of this research. As Shuster and Oh point out:

> A recurrent problem in spacecraft attitude determination is to determine the attitude from a set of vector measurements. Thus, an orthogonal matrix *A* (the attitude matrix or direction-cosine matrix) is sought which satisfies
>
> $$A\hat{V}_i = \hat{W}_i \tag{9.1}$$
>
> where $\hat{V}_1,\ldots,\hat{V}_n$ are a set of reference unit vectors, which are *n* known directions (e.g., the direction of the Earth, the Sun, a star, or the geomagnetic field) in the reference coordinate system, and $\hat{W}_1,\ldots,\hat{W}_n$ are the observation unit vectors which are the same *n* directions as measured in the spacecraft-body coordinate system...Because both the observation and the reference unit vectors are corrupted by error, a solution for *A* does not exist in general, not even for $n = 2$ [35].

While many approaches have been researched to better solve this problem, the intent here is to leverage promising sensor fusion and filtering techniques to devise an attitude determination algorithm suitable for implementation using low-cost sensors. As pointed out in the introduction, many techniques have been devised and successfully used for applications where large budgets are available. The low-cost approach is accompanied by a new set of challenges, most stemming from the lower accuracy and lesser number of sensors available when cost is limited. Here, the focus is on what can be done with less, thereby providing a useful algorithm to the experimenter with limited resources.

9.2 Overview of the Approach

As supported in previous chapters, the traditional approach to this type of estimation problem is based on the Extended Kalman Filter (EKF). As a baseline for this study, such a filter is designed and tested. One difference from what might be considered the "standard" approach is the use of the Gauss-Newton error minimization routine to reduce the complexity of the state equations used to calculate the predicted observation at the next time step. A similar approach was used by Marins for his low dynamic vehicle using gravity (accelerometer), magnetic field, and rotational rate sensors [11]. Using this approach, four derived measurements replace six actual measured quantities. A second, and perhaps more significant departure, is the design and test of an algorithm using the Unscented Kalman Filter (UKF) in place of the EKF. This second algorithm still uses the Gauss-Newton approach to simplify the state equations, but takes advantage of the superior nonlinear estimation characteristics of the UKF as compared to the EKF. Additionally, the feasibility of eliminating the rate sensors in order to further reduce system cost, is investigated. Both the EKF-based and the UKF-based algorithms are tested with and without rate sensors and the comparative performance is analyzed.

9.2.1 Measurements Available to the Algorithm

As developed in Chapter 6, the sensors making up the sensor suite for this work consist of a minimum of a Sun sensor and a magnetometer. Depending upon the implementation, gyroscopes are included to measure rates in the body frame. The detailed characteristics of the sensors used are discussed in a later section. This section concentrates on the type and number of measurements to be fed to the algorithm. I will distinguish between the word "measurement" meaning something coming out of a sensor, and the word "observation" meaning something passed to a filtering algorithm for it to operate on. These terms are often used interchangeably in the literature, but this definition will distinguish between them here.

The measurements available from the system include three orthogonal components of the Sun vector, three orthogonal components of the magnetic field vector, and three orthogonal components of the rotational rate vector. Each of these vectors may be expressed either in the inertial frame, or in the body frame of the rotating object. For this implementation, the Sun and magnetic field vector are assumed to be available from models in the inertial frame, and are measured by sensors in the body frame. The rotational rates are handled differently, depending upon whether rate sensors are included in the sensor suite or if the rates are derived by differencing quaternions to get quaternion rates and then body rates. In either case, the rotational rates are expressed as body rates about the body principle axes.

The observations that are fed to the EKF or UKF portions of the algorithms consist of seven elements. The Sun and magnetic field vectors are processed by the Gauss-Newton routine to provide a best estimate of the attitude quaternion. The four components of this quaternion are the "observations" upon which the filters operate, not the measured vector components. The remaining three observations are the three elements of the rotational rate vector, whether measured or derived by differencing.

It is important to consider the filter time step required for successful estimation of the rocket rotational motion. Especially in light of the rudimentary dynamic model used by both filter types to propagate the motion forward one time step, this time step must be small relative to the time constants of the expected motion. The highest frequency motion simulated here is the 225 revolution per minute (rev/min) motion about the longitudinal axis of the rocket. This is equal to approximately 23.6 rad/sec which implies a time scale in the neighborhood of .04 seconds. If a Nyquist guideline of sampling at greater than or equal to three times the highest frequency of the signal is used, then the filter time step should be no greater than .013 seconds to accurately capture the motion. In the simulation portion of this work, to be detailed later, data points are simulated at a frequency at least three times the highest frequency of the expected motion. Likewise, a filter time step less than or equal to one-third the shortest time constant is used. In recognition of the goal to be able to implement the final algorithm in real time, for the baseline case where the maximum rate of rotation is 225 rev/min, filtering is accomplished at a time step equal to .01 seconds. Results in Chapter 11 illustrate the negative performance impact of filtering at a longer time step than that predicted by this approach.

9.2.2 Processing Methods

Essentially, four algorithms are developed for this work. They may be described as Gauss-Newton / EKF / no rate sensors, Gauss-Newton / EKF / with rates sensors, Gauss-Newton / UKF / no rate sensors, and Gauss-Newton / UKF / with rate sensors. In each case, the Sun and magnetic field vectors are measured in the body frame. At the same time their inertial frame values are determined by accessing models with an accurate time and position. The Gauss-Newton routine minimizes an error function to determine the best-fit transformation between the two coordinate frames and represents this as an estimate of the attitude quaternion. Next, the four components of this quaternion, along with the three components of the rotational rate vector, are provided as observations to the EKF or the UKF as the case may be. This filtering algorithm then uses the observations, an embedded dynamic model, and a "memory" of past state vectors to produce an updated estimate of the rocket state vector. Being a sequential filter, at each step only the current estimate is maintained, as opposed to a batch method that requires all past estimates to be stored for processing [85]. In cases where rotational rate measurements are provided by a sensor, the measurements in each axis are fed to the filters as observations. In cases where rates are not measured, there is a one step time delay to allow for a differencing of quaternion components over one time step. This differencing produces a derived quaternion rate according to the following equation

$$\dot{q} = \frac{q_k - q_{k-1}}{\Delta t} \tag{9.2}$$

where *k* and *k-1* signify successive time steps and Δt signifies the magnitude of the time step. Here each of the four components is differenced to provide a four element time rate of change of the quaternion. This quaternion rate is then used to calculate the three body rates according to

$$\omega_i = 2\sum_{j=1}^{4} q_{ij}\dot{q}_j \tag{9.3}$$

obtained by rearranging equation (4.22). This effectively provides a noisy "measurement" of the angular rates expressed in the body frame. These in turn, may be fed to the filtering portion of the algorithm as an observation of the rates in the case where no true rate sensor is employed.

9.2.3 Block Diagram of Overall Process

Figure 9.1 provides a visual summary of the overall process. The following sections will fill in the detail of how the various blocks are implemented. Tuning of the filters, or choosing the various parameters, will be discussed in Chapter 10. A discussion of the detailed results follows in Chapter 11.

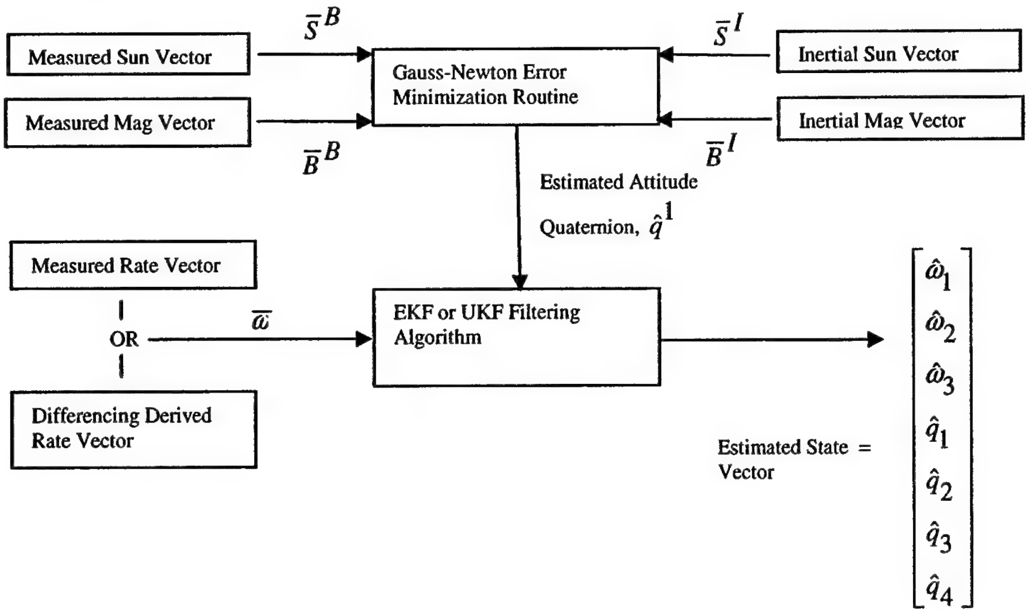

Measured Sun Vector
$\bar{S}^B$
Gauss-Newton Error Minimization Routine
$\bar{S}^I$
Inertial Sun Vector
Measured Mag Vector
$\bar{B}^B$
$\bar{B}^I$
Inertial Mag Vector
Estimated Attitude Quaternion, $\hat{q}^1$
Measured Rate Vector
OR
$\bar{\omega}$
EKF or UKF Filtering Algorithm
Differencing Derived Rate Vector
Estimated State Vector =
$\hat{\omega}_1$
$\hat{\omega}_2$
$\hat{\omega}_3$
$\hat{q}_1$
$\hat{q}_2$
$\hat{q}_3$
$\hat{q}_4$

Figure 9.1: Overview of Attitude Estimation Algorithm

9.3 Sensor Characteristics

The relative merits of the various attitude sensors available were discussed in Chapter 6. For the development of this algorithm, the Sun vector and the magnetic field vector are used as references. The following sections describe the sensor characteristics assumed for the development.

9.3.1 Sun Sensor

Based on the reported accuracy from the SRP-4 mission, the Tokai University designed and built Sun sensor is capable of +/-4 degrees measurement accuracy [53]. Since this sensor makes use of relatively straightforward technology and manufacturing techniques, this accuracy should be typical of what can be achieved by a low-cost program. Therefore, this development assumes a standard deviation, in each of the three body axes, of 1.333 degrees. In order to achieve this, more than one sensor is required in order to measure the angle in each of the body axes. Additionally, due to view angle considerations generated by the spinning motion inherent in sounding rockets, additional sensors may be required to provide measurements at the frequency determined by the necessary filter time step. This is a hardware design issue beyond the scope of this effort.

9.3.2 Magnetometer

As detailed in Chapter 6, a measurement accuracy of 10 degrees should be easily attainable, even in a low-cost endeavor. As developed in Chapter 6, this corresponds to a measurement standard deviation of 3.333 degrees. This will be the baseline measurement standard deviation used for the algorithm design. Here, as with the Sun sensor, the sensor must measure in each of the body frame axes, and the measurement interval is assumed to be .01 seconds. Among other things, the hardware design would need to take into account sampling rate as well as spin rate of the vehicle. For a magnetometer, unlike for the Sun sensor, view angles will not typically be an issue.

9.3.3 Rate Gyroscopes

The baseline rate gyroscopes used to develop this algorithm have a sampling, or measurement, rate of 100 samples per second. The measurement standard deviation, in each body frame axis, is assumed to be 1 revolution per minute. This is a very coarse rate measurement, and is in line with what can be expected from the low-cost MEMS gyroscopes now available. The rate measurements described in [11] have a much lower standard deviation, on the order of 7.4×10^{-5} to .03 rad/sec, or .001 to .3 rev/min. Therefore, the 1 rev/min measurement accuracy corresponding to a 0.333 rev/min should be easily attainable, and is the baseline for this effort.

9.4 Dynamic Modeling of Rocket Motion

As discussed in Chapter 5, a dynamic model is required for embedding in the EKF and UKF routines. This model is not a high fidelity model for accurate propagation of the rocket motion, but a coarse model for propagating individual states from one time step to the next as part of the estimation process.

9.4.1 Simplified Dynamic Model

Based on the rationale laid out in Chapter 5, a simplified dynamic model is used for development of this algorithm. Figure 5.1, depicting the process by which rotational rates are driven by white noise, is repeated here as Figure 9.2 as an aid to the reader. As discussed earlier, this model is embedded in the filter algorithm and is used to propagate the rotational rate states forward one time step. Its suitability depends upon the assumption that the inertia of the physical rocket prevents radical changes of rocket motion over the duration of one filter time step. As detailed in Section 9.2.1, the filter time step used for this analysis is .01 seconds. This step size is less than one third the time scale of the highest frequency of the expected rocket motion. This highest expected frequency is the 225 rev/min or 23.6 rad/sec spin rate about the longitudinal axis, corresponding to a time scale of approximately .013 sec. Again, in this figure, ω is a 3×1 vector of angular rates ω_1, ω_2, and ω_3 in the body frame, w_r is a 3×1 vector of white noise components generating ω_1, ω_2, and ω_3, and τ_r is a 3×1 vector of time constants corresponding to ω_1, ω_2, and ω_3.

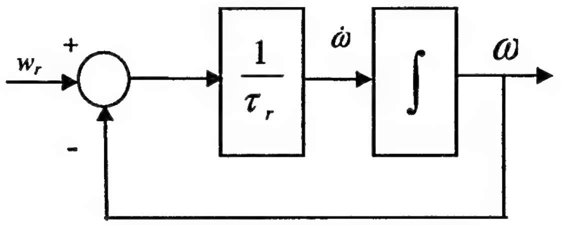


Figure 9.2: White Noise Driven Angular Rates (After Figure 3.1 in [11])

9.4.2 Model Parameters

There are very few parameters necessary to implement this simplified model. The spectral characteristics of the white noise, embodied in w_r, must be defined. Additionally, the time constants, τ_r, must be determined. For this effort, both are solved for empirically and this process will be described in detail in Chapter 10 during the discussion of filter tuning. Clearly, such a model lacks a measure of fidelity, but has the great advantage of simplicity, and transportability between systems. None of the parameters is unique to a particular rocket, but is instead a function of the motion itself. While not suitable for detailed and accurate prediction of motion, it is sufficient for embedding in the EKF and the UKF for state propagation across small time steps. In this respect, the benefits of simplicity and transportability outweigh what is lost in terms of model fidelity.

9.5 Quaternion-Based Attitude

The attitude estimation algorithms developed here rely on quaternions for their attitude representation. There are numerous advantages to using this parameterization, many of which were outlined earlier in Chapter 4. The principle motivation for this application is avoidance of singularity issues and minimization of the number of calculations. As illustrated in Figure 9.1, the Gauss-Newton routine produces the first estimate of the attitude quaternion by performing an error minimization. This quaternion estimate is normalized to magnitude of 1, to insure that it represents a pure rotation. The mechanics of the Gauss-Newton routine are described in a later section. This initial estimate of the quaternion is then input to the filtering routine, either EKF or UKF-based, as a "measurement" where it is further refined into a better estimate of the attitude transformation. Several of the properties of quaternions are utilized both in the simulation step of this work as well as in the "differencing" process to generate rotational rate "measurements" when rate sensors are not used. While the overall use of quaternions is addressed in Chapter 4, the next two sections highlight those properties most critical to the implementation here.

9.5.1 Relationship Between Quaternions and Euler Angles

For reasons to be explained in Section 10.1.1, the simulation of data for developing and testing the algorithm relies on propagating Euler angles and then converting these angles to a quaternion. Therefore the relationship between the two methods of representing attitude is an important aspect of this effort. This key result, first presented in Chapter 4, (4.27) is repeated here as an aid to the reader.

$$\begin{aligned} q_1 &= \cos\frac{\psi}{2}\sin\frac{\theta}{2}\cos\frac{\phi}{2} + \sin\frac{\psi}{2}\sin\frac{\theta}{2}\sin\frac{\phi}{2} \\ q_2 &= -\cos\frac{\psi}{2}\sin\frac{\theta}{2}\sin\frac{\phi}{2} + \sin\frac{\psi}{2}\sin\frac{\theta}{2}\cos\frac{\phi}{2} \\ q_3 &= \cos\frac{\psi}{2}\cos\frac{\theta}{2}\sin\frac{\phi}{2} + \sin\frac{\psi}{2}\cos\frac{\theta}{2}\cos\frac{\phi}{2} \\ q_4 &= \cos\frac{\psi}{2}\cos\frac{\theta}{2}\cos\frac{\phi}{2} - \sin\frac{\psi}{2}\cos\frac{\theta}{2}\sin\frac{\phi}{2} \end{aligned} \quad . \tag{9.4}$$

In addition to this relationship between the Euler angles and the attitude quaternion components, the relationship between body rates and quaternion rates is important in terms of generating rotational rates to input to the filtering algorithm when rate sensors are not used.

9.5.2 Relationship Between Quaternion Rates and Body Rates

For the case where rotational rates are not directly measured, an alternative method relies on deriving the rotational rates by differencing subsequent quaternions, and then converting the resulting quaternion "rate" into a body-frame rotational rate vector. In order to do this, one must have a relationship between the two. As presented in Chapter 4 as (4.22), this relationship is

$$\frac{d}{dt}(q_j) = \dot{q}_j = \frac{1}{2}\sum_{i=1}^{4} q_{ij}\omega_i \,. \tag{9.5}$$

where q_j are the components of the attitude quaternion and ω_i are the components of the body-frame rotational rate vector. The mechanics of deriving rate measurements were presented in Section 9.2. Another way of looking at this relationship between body rates and quaternions is to combine the simplified dynamics model of Section 9.3, the implementation of equation (4.22) shown above, and the normalization step. Again, the normalization is necessary to insure that the quaternion produces a pure rotation, with no change in magnitude. Figure 9.3 below is a pictorial of the resulting process model which is used to form the state equations that are integrated to propagate the estimated states forward one time step.

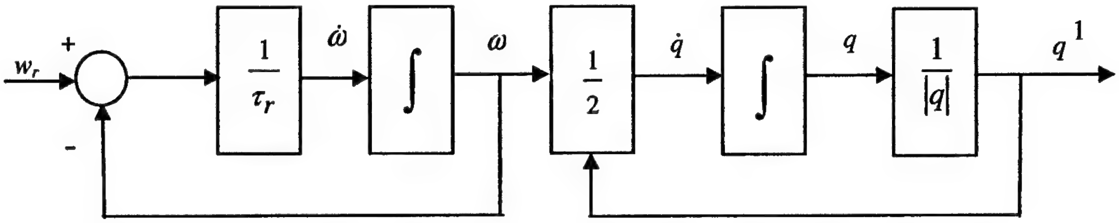


Figure 9.3: Process Model Relating Unit Quaternion to White Noise Generated Angular Rates

9.6 Gauss-Newton Parameter Optimization

As part of this algorithm to form a best estimate of attitude motion, a Gauss-Newton error minimization is used to find the best transformation between the inertial frame and the rotated body frame. Since an attitude quaternion is a representation of this transformation, minimizing the error in the estimated transformation essentially provides the best estimate of the attitude quaternion. As demonstrated by Marins [11], implementing such a routine effectively reduces the complexity of the state equations for a filtering routine. A traditional approach for implementing a Kalman filter-based algorithm would likely start with the formation of a state vector made up of three rotational rates, and then either four quaternion components, three Euler angles, or nine elements of a direction cosine matrix to fully describe the rotational motion of the rocket. Within the EKF, state equations must be formed to allow propagation of the states forward. These are typically functions of the current state vector. In a traditional formulation, the measured values of the rates and the components of the two reference vectors would be the observations. In this case, once the states were propagated forward one time step, the predicted observations at that time step would need to be calculated from the propagated states. As derived by Marins [11], these functions are complex, nonlinear expressions that do not lend themselves to the real-time application sought here. As will be seen in a later section, using the Gauss-Newton routine to generate derived "measurements" of the attitude quaternion greatly simplifies these state equations, and the generation of the predicted observations. The outputs, which are compared to measurements to form residuals, are typically some function of the measured states. Using an error minimization technique, in this case Gauss-Newton error minimization, to determine an attitude quaternion allows the formation of output equations that are identical to the measured rotational rates and the quaternion components, thereby reducing the complexity of the filter algorithm. This not only simplifies the filter design, but also yields a corresponding reduction in the number of calculations and hence run time required to execute the algorithm.

9.6.1 Design of the Gauss-Newton Routine

Similar to the development in [11], the Gauss-Newton routine is used here to reduce the dimension of the state vector. Essentially, instead of using the measured values of the Sun and magnetic field vectors as "observations" for the filtering algorithm, Gauss-Newton is used to generate an attitude quaternion which serves as a "derived observation." This condenses the six components of the measured reference vectors into four elements of the quaternion. This process requires the Sun and magnetic field vectors to be known in the inertial frame for comparison to those measured in the body frame of the rocket. This assumes the availability of an accurate solar ephemeris, an accurate magnetic field model, a good time reference, and an accurate position of the rocket. For this work, these values are simulated as part of the controlled evaluation of the algorithms designed. For a real-world implementation, a radar or GPS position might be used along with a time reference to access models and determine inertial Sun and magnetic field vectors at each time step.

Based on this general concept, in conjunction with the Gauss-Newton equations laid out in Chapter 7, the detailed design may be derived. The premise upon which the choice of Gauss-Newton is based is that there exists a quadratic error function that can be minimized. Here we have two three component vectors, the Sun vector and the magnetic field vector, that are measured in the body frame and known in the inertial frame. The two known vectors can be concatenated into a single 6×1 reference vector. Likewise, the two measured vectors can be formed into a single 6×1 measurement vector. The goal is to find the transformation that takes one into the other. If *M(q)* is related to *R(q)*, the transformation between the body frame and the inertial frame, as

$$M(q) = \begin{bmatrix} R(q) & 0 \\ 0 & R(q) \end{bmatrix} \tag{9.6}$$

then the error vector can be formed as [11]

$$\bar{\varepsilon}(q) = \bar{y}_k^I - M(q)\bar{y}_k^B \tag{9.7}$$

where

$\bar{\varepsilon}(q)$ = error vector (dimension 6×1),

$\bar{y}_k^I$ = Sun and magnetic field reference vector in inertial frame at time step *k* (dimension 6×1),

$\bar{y}_k^B$ = Sun and magnetic field reference vector measured in body frame at time step *k* (dimension 6×1),

$M(q)$ = transformation matrix composed from R, the 3x3 rotation matrix relating body frame to inertial reference (dimension 6×6).

For this problem the Jacobian described by equation (7.10) using N=1 and n=4 becomes

$$J_k = \left[\frac{\partial(\bar{y}_k^I - M(q)\bar{y}_k^B)}{\partial \hat{q}_{1k}} \quad \frac{\partial(\bar{y}_k^I - M(q)\bar{y}_k^B)}{\partial \hat{q}_{2k}} \quad \frac{\partial(\bar{y}_k^I - M(q)\bar{y}_k^B)}{\partial \hat{q}_{3k}} \quad \frac{\partial(\bar{y}_k^I - M(q)\bar{y}_k^B)}{\partial \hat{q}_{4k}} \right]. \tag{9.8}$$

$\bar{y}_k^I$, the reference vector composed by concatenating the inertial Sun and magnetic field vectors, does not depend on the quaternion elements. Therefore, J_k reduces to

$$J_k = -\left[\frac{\partial M(q)\bar{y}_k^B}{\partial \hat{q}_{1k}} \quad \frac{\partial M(q)\bar{y}_k^B}{\partial \hat{q}_{2k}} \quad \frac{\partial M(q)\bar{y}_k^B}{\partial \hat{q}_{3k}} \quad \frac{\partial M(q)\bar{y}_k^B}{\partial \hat{q}_{4k}} \right] \tag{9.9}$$

which is dimension 6×4. In Chapter 4, equation (4.10), we saw that the rotation matrix *R*, relating the rotated body frame to the inertial frame, is defined as

$$R(q) = \begin{bmatrix} (q_1^2 - q_2^2 - q_3^2 + q_4^2) & 2(q_1q_2 + q_3q_4) & 2(q_1q_3 - q_2q_4) \\ 2(q_1q_2 - q_3q_4) & (-q_1^2 + q_2^2 - q_3^2 + q_4^2) & 2(q_1q_4 + q_2q_3) \\ 2(q_1q_3 + q_2q_4) & 2(-q_1q_4 + q_2q_3) & (-q_1^2 - q_2^2 + q_3^2 + q_4^2) \end{bmatrix}. \tag{9.10}$$

Substituting *R(q)* into equation (9.6) and multiplying yields the 6×1 matrix

$$M(q)\bar{y}_k^B = \begin{bmatrix} R_{11}y_1^B + R_{12}y_2^B + R_{13}y_3^B \\ R_{21}y_1^B + R_{22}y_2^B + R_{23}y_3^B \\ R_{31}y_1^B + R_{32}y_2^B + R_{33}y_3^B \\ R_{11}y_4^B + R_{12}y_5^B + R_{13}y_6^B \\ R_{21}y_4^B + R_{22}y_5^B + R_{23}y_6^B \\ R_{31}y_4^B + R_{32}y_5^B + R_{33}y_6^B \end{bmatrix} \tag{9.11}$$

and taking the first partial as indicated in equation (9.9), one gets

$$\frac{\partial M(q)\bar{y}_k^B}{\partial \hat{q}_1} = \begin{bmatrix} 2\hat{q}_1 y_1^B + 2\hat{q}_2 y_2^B + 2\hat{q}_3 y_3^B \\ 2\hat{q}_2 y_1^B - 2\hat{q}_1 y_2^B - 2\hat{q}_4 y_3^B \\ 2\hat{q}_3 y_1^B + 2\hat{q}_4 y_2^B - 2\hat{q}_1 y_3^B \\ 2\hat{q}_1 y_4^B + 2\hat{q}_2 y_5^B + 2\hat{q}_3 y_6^B \\ 2\hat{q}_2 y_4^B - 2\hat{q}_1 y_5^B - 2\hat{q}_4 y_6^B \\ 2\hat{q}_3 y_4^B + 2\hat{q}_4 y_5^B - 2\hat{q}_1 y_6^B \end{bmatrix} = 2\begin{bmatrix} \hat{q}_1 & \hat{q}_2 & \hat{q}_3 & 0 & 0 & 0 \\ \hat{q}_2 & -\hat{q}_1 & -\hat{q}_4 & 0 & 0 & 0 \\ \hat{q}_3 & -\hat{q}_4 & -\hat{q}_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \hat{q}_1 & \hat{q}_2 & \hat{q}_3 \\ 0 & 0 & 0 & \hat{q}_2 & -\hat{q}_1 & -\hat{q}_4 \\ 0 & 0 & 0 & \hat{q}_3 & -\hat{q}_4 & -\hat{q}_1 \end{bmatrix} \begin{bmatrix} y_1^B \\ y_2^B \\ y_3^B \\ y_4^B \\ y_5^B \\ y_6^B \end{bmatrix}. \tag{9.12}$$

Similarly, the remaining partials to populate J_k are

$$\frac{\partial M(q)\bar{y}_k^B}{\partial \hat{q}_2} = \begin{bmatrix} -2\hat{q}_2 y_1^B + 2\hat{q}_1 y_2^B + 2\hat{q}_4 y_3^B \\ 2\hat{q}_1 y_1^B + 2\hat{q}_2 y_2^B + 2\hat{q}_3 y_3^B \\ -2\hat{q}_4 y_1^B + 2\hat{q}_3 y_2^B - 2\hat{q}_2 y_3^B \\ -2\hat{q}_2 y_4^B + 2\hat{q}_1 y_5^B + 2\hat{q}_4 y_6^B \\ 2\hat{q}_1 y_4^B + 2\hat{q}_2 y_5^B + 2\hat{q}_3 y_6^B \\ -2\hat{q}_4 y_4^B + 2\hat{q}_3 y_5^B - 2\hat{q}_2 y_6^B \end{bmatrix} = 2\begin{bmatrix} -\hat{q}_2 & \hat{q}_1 & \hat{q}_4 & 0 & 0 & 0 \\ \hat{q}_1 & \hat{q}_2 & \hat{q}_3 & 0 & 0 & 0 \\ -\hat{q}_4 & \hat{q}_3 & -\hat{q}_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & -\hat{q}_2 & \hat{q}_1 & \hat{q}_4 \\ 0 & 0 & 0 & \hat{q}_1 & \hat{q}_2 & \hat{q}_3 \\ 0 & 0 & 0 & -\hat{q}_4 & \hat{q}_3 & -\hat{q}_2 \end{bmatrix} \begin{bmatrix} y_1^B \\ y_2^B \\ y_3^B \\ y_4^B \\ y_5^B \\ y_6^B \end{bmatrix} \tag{9.13}$$

$$\frac{\partial M(q)\bar{y}_k^B}{\partial \hat{q}_3} = \begin{bmatrix} -2\hat{q}_3 y_1^B - 2\hat{q}_4 y_2^B + 2\hat{q}_1 y_3^B \\ 2\hat{q}_4 y_1^B - 2\hat{q}_3 y_2^B + 2\hat{q}_2 y_3^B \\ 2\hat{q}_1 y_1^B + 2\hat{q}_2 y_2^B + 2\hat{q}_3 y_3^B \\ -2\hat{q}_3 y_4^B - 2\hat{q}_4 y_5^B + 2\hat{q}_1 y_6^B \\ 2\hat{q}_4 y_4^B - 2\hat{q}_3 y_5^B + 2\hat{q}_2 y_6^B \\ 2\hat{q}_1 y_4^B + 2\hat{q}_2 y_5^B + 2\hat{q}_3 y_6^B \end{bmatrix} = 2\begin{bmatrix} -\hat{q}_3 & -\hat{q}_4 & \hat{q}_1 & 0 & 0 & 0 \\ \hat{q}_4 & -\hat{q}_3 & \hat{q}_2 & 0 & 0 & 0 \\ \hat{q}_1 & \hat{q}_2 & \hat{q}_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -\hat{q}_3 & -\hat{q}_4 & \hat{q}_1 \\ 0 & 0 & 0 & \hat{q}_4 & -\hat{q}_3 & \hat{q}_2 \\ 0 & 0 & 0 & \hat{q}_1 & \hat{q}_2 & \hat{q}_3 \end{bmatrix} \begin{bmatrix} y_1^B \\ y_2^B \\ y_3^B \\ y_4^B \\ y_5^B \\ y_6^B \end{bmatrix} \tag{9.14}$$

$$\frac{\partial M(q)\bar{y}_k^B}{\partial \hat{q}_4} = \begin{bmatrix} 2\hat{q}_4 y_1^B - 2\hat{q}_3 y_2^B + 2\hat{q}_2 y_3^B \\ 2\hat{q}_3 y_1^B + 2\hat{q}_4 y_2^B - 2\hat{q}_1 y_3^B \\ -2\hat{q}_2 y_1^B + 2\hat{q}_1 y_2^B + 2\hat{q}_4 y_3^B \\ 2\hat{q}_4 y_4^B - 2\hat{q}_3 y_5^B + 2\hat{q}_2 y_6^B \\ 2\hat{q}_3 y_4^B + 2\hat{q}_4 y_5^B - 2\hat{q}_1 y_6^B \\ -2\hat{q}_2 y_4^B + 2\hat{q}_1 y_5^B + 2\hat{q}_4 y_6^B \end{bmatrix} = 2\begin{bmatrix} \hat{q}_4 & -\hat{q}_3 & \hat{q}_2 & 0 & 0 & 0 \\ \hat{q}_3 & \hat{q}_4 & -\hat{q}_1 & 0 & 0 & 0 \\ -\hat{q}_2 & \hat{q}_1 & \hat{q}_4 & 0 & 0 & 0 \\ 0 & 0 & 0 & -\hat{q}_4 & \hat{q}_3 & \hat{q}_2 \\ 0 & 0 & 0 & \hat{q}_3 & -\hat{q}_4 & -\hat{q}_1 \\ 0 & 0 & 0 & \hat{q}_2 & -\hat{q}_1 & \hat{q}_4 \end{bmatrix} \begin{bmatrix} y_1^B \\ y_2^B \\ y_3^B \\ y_4^B \\ y_5^B \\ y_6^B \end{bmatrix}. \tag{9.15}$$

Substituting these partials into equation (9.9) to form the Jacobian and then substituting the expression for the error vector from equation (9.7) into equation (7.15) yields

$$\hat{q}_{k+1} = \hat{q}_k - [J^T(\hat{q}_k)J(\hat{q}_k)]^{-1} J^T(\hat{q}_k)(\bar{y}_k^I - M(\hat{q})\bar{y}_k^B) \tag{9.16}$$

This equation is supplied with the last estimated quaternion as an initial guess and then iterated until the convergence criterion is met or until the maximum number of iterations is reached. In this implementation the convergence criteria is the mean square error as determined from the error vector described by equation (9.7), and is shown below

$$F(q) = \bar{\varepsilon}(q)^T \bar{\varepsilon}(q) = (\bar{y}_k^N - M(q)\bar{y}_k^B)^T (\bar{y}_k^N - M(q)\bar{y}_k^B) \tag{9.17}$$

where

$F(q)_k$ = error function to be minimized,

$\bar{\varepsilon}(q)$ = error vector,

$\bar{y}_k^N$ = measurement vector in the "navigation" frame,

$M(q)$ = transformation matrix from "body" to "navigation" frame,

$\bar{y}_k^B$ = measurement vector in the "body" frame.

The output is the next Gauss-Newton estimated attitude quaternion based on minimizing the error in the transformation from the inertial frame to the body frame. The software implementing this routine is gauss_newton.m and is found in Appendix A.

9.6.2 Inputs to Error Minimization Routine

Given the description of the detailed routine design provided in the last section, Figure 9.4 below outlines the general process for determining the attitude quaternion estimate.

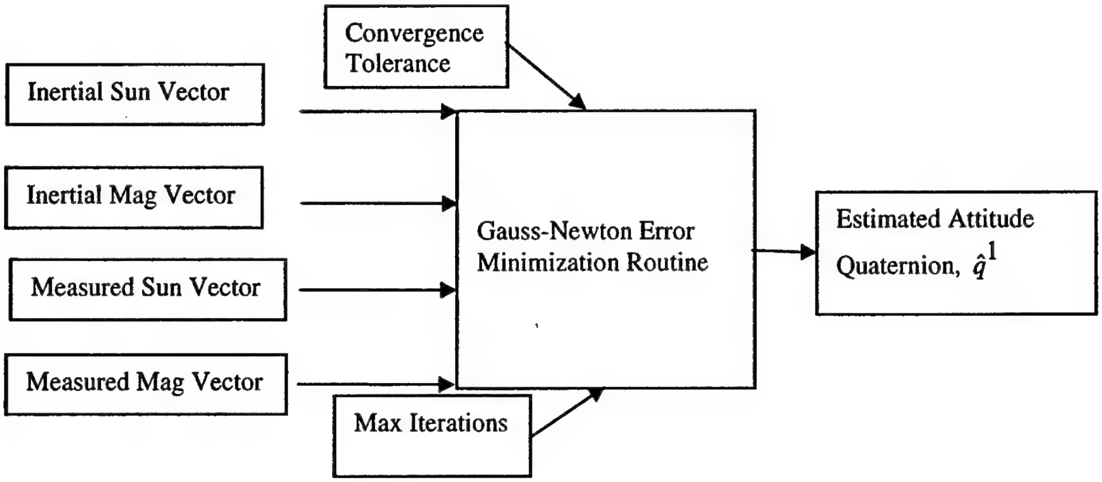

Convergence Tolerance
Inertial Sun Vector
Inertial Mag Vector
Measured Sun Vector
Measured Mag Vector
Gauss-Newton Error Minimization Routine
Estimated Attitude Quaternion, $\hat{q}^1$
Max Iterations

Figure 9.4: Outline of Gauss-Newton Routine

This diagram illustrates the inputs to the routine, and shows the output to be the estimated attitude quaternion. As will be demonstrated later, this output, along with the three measured rotational rates, comprise the observations for the filtering algorithm. This concludes the discussion of the "front end" of the algorithm that is common to both the EKF and UKF based algorithms. The next section details the development of the traditional EKF based algorithm.

9.7 Extended Kalman Filter Design

Having discussed what inputs are available to the filtering routine and how the Gauss-Newton routine provides a derived measurement, the elements are now in place to look at the detailed design of the various filtering algorithms. This section addresses the traditional Extended Kalman Filter (EKF) approach. While the theory and characteristics have been covered in previous chapters, here the detailed design is reviewed. The equations for the various matrices can be found in any one of many excellent references on Kalman filtering [17], [68], [71]. Figure 9.5 provides an overview of the entire EKF structure as a prelude to how the filter is implemented to include what matrices and other quantities must be defined.

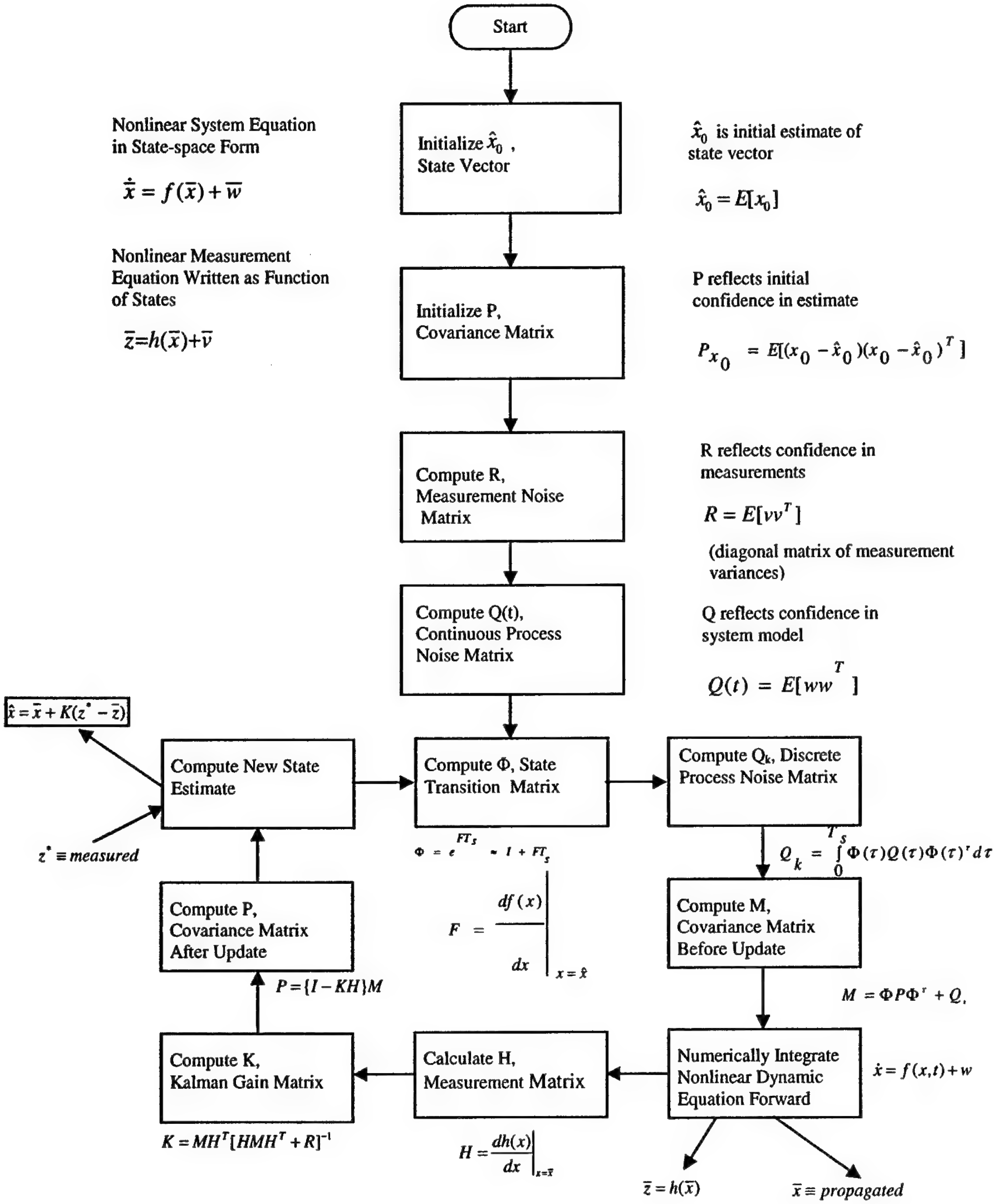

Start
Nonlinear System Equation in State-space Form
$\dot{\bar{x}} = f(\bar{x}) + \bar{w}$
Initialize $\hat{x}_0$, State Vector
$\hat{x}_0$ is initial estimate of state vector
$\hat{x}_0 = E[x_0]$
Nonlinear Measurement Equation Written as Function of States
$\bar{z}=h(\bar{x})+\bar{v}$
Initialize P, Covariance Matrix
P reflects initial confidence in estimate
$P_{x_0} = E[(x_0 - \hat{x}_0)(x_0 - \hat{x}_0)^T]$
Compute R, Measurement Noise Matrix
R reflects confidence in measurements
$R = E[vv^T]$
(diagonal matrix of measurement variances)
Compute Q(t), Continuous Process Noise Matrix
Q reflects confidence in system model
$Q(t) = E[ww^T]$
$\hat{x} = \bar{x} + K(z^* - \bar{z})$
Compute New State Estimate
$z^* \equiv measured$
Compute Φ, State Transition Matrix
Compute Q_k, Discrete Process Noise Matrix
$\Phi = e^{FT_s} \approx I + FT_s$
$F = \left.\frac{df(x)}{dx}\right|_{x=\hat{x}}$
$Q_k = \int_0^{T_s} \Phi(\tau)Q(\tau)\Phi(\tau)^T d\tau$
Compute P, Covariance Matrix After Update
$P = \{I - KH\}M$
Compute M, Covariance Matrix Before Update
$M = \Phi P \Phi^T + Q_k$
Compute K, Kalman Gain Matrix
$K = MH^T[HMH^T + R]^{-1}$
Calculate H, Measurement Matrix
$H = \left.\frac{dh(x)}{dx}\right|_{x=\bar{x}}$
Numerically Integrate Nonlinear Dynamic Equation Forward
$\dot{x} = f(x,t) + w$
$\bar{z} = h(\bar{x})$
$\bar{x} \equiv propagated$

Figure 9.5: Overall Block Diagram of Extended Kalman Filter

9.7.1 Choice of State Vector

Designing the filtering algorithm requires choosing what states will be estimated. Since the goal is to estimate the attitude of the rocket, things that directly impact its rotational motion over time should be included. As discussed throughout the earlier chapters, the attitude of the vehicle will be parameterized using quaternions. Therefore, it makes sense to include the four quaternion elements as states. Additionally, since the attitude is not expected to be a static orientation, the rotational rates are included as states to be estimated. The state vector is represented as

$$\bar{x} = \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ q_1 \\ q_2 \\ q_3 \\ q_4 \end{bmatrix}. \tag{9.18}$$

9.7.2 Derivation and Definition of Filter Matrices

As depicted in Figure 9.5, there are a large number of matrices that must be defined for implementation of the Extended Kalman Filter (EKF). The software implementation is found in Appendix A. The two files rates_ekf.m and norates_ekf.m implement the EKF routines for cases where rotational rate measurements come from sensors and where rotational rate "measurements" come from differencing, respectively. This section details the derivation of the various equations and matrices coded in these routines.

9.7.2.1 Initializing the State Vector, $\hat{\mathbf{x}}_0$

The EKF filter routine requires an initial guess, or estimate, of the state vector to begin processing. While a simple solution is to begin with all zeros, this risks exacerbating the well-known issues concerning filter divergence due to poor initialization. Realistically, at the initial time, a reasonable estimate of the rocket attitude and rates should be available. Since this work relies on a simulation environment, the user is prompted for the initial conditions, or the program defaults to the actual initial conditions with a user defined percent error. For the results in this work, a five percent error was added to the actual initial values coming from the simulation algorithm, and these were used to initialize the state vector. Since the initial simulated rates are zero, the error added is zero. The quaternion component errors are not zero. This is realistic as the states of a rocket on the launch rail should be relatively easily estimated, with less error in the rates than in the quaternion elements. In a real world implementation, if a filter "restart" was required during a flight, the snapshot provided by the Gauss-Newton routine could be used to reinitialize the filter. While divergence may occur for a very poor initialization, anything close to the actual orientation and rates should allow for proper operation.

9.7.2.2 Initializing the Covariance Matrix, *P*

The covariance matrix is a measure of the confidence in the estimate. In this case the initial covariance matrix reflects the confidence in the initial estimate of the state vector. This matrix is defined as

$$P_{x_0} = E[(x_0 - \hat{x}_0)(x_0 - \hat{x}_0)^T] \tag{9.19}$$

where E symbolizes the "expectation" operator and this may be interpreted as the squared error in the initial estimate. In terms of the program implemented here, this matrix is initialized with the user-defined confidence in the initial estimate, either based on the user-defined error to the actual initial conditions, or the entered confidence in the user-entered initial conditions. For software implementation, this matrix is generated as a diagonal matrix with the squares of the error corresponding to each initial state along the diagonal. For the study conducted here, this error in the initial states is always five percent of the actual initial condition as described in the previous section Since this is a covariance matrix representing squared error, it must be positive definite by definition. Therefore, a small positive value is added to each element to insure the matrix remains positive definite in the case where the error in an initial estimated state is deemed to be zero.

9.7.2.3 Computing the Measurement Noise Matrix, *R*

The Measurement Noise Matrix, R, gives a reflection of the confidence in the accuracy of the measurements. It too is a covariance matrix. It is defined as

$$R = E[vv^T] \tag{9.20}$$

where the symbol E is again the expectation operator and v is the vector seen in (8.2) representing the white noise process that corrupts the sensor measurements. Taking the expectation shown above, this matrix may be interpreted as a diagonal matrix with the squares of the sensor standard deviation, or their variances, along the diagonal. In the non-traditional implementation used here, however, there are actually nine sensor measurements coming from the three rates and the three components from each reference vector. The Gauss-Newton routine condenses these last six into four "measurements." As a result, the dimension of R must be reduced from 9x9 to 7x7 in order to be compatible for future matrix multiplication operations. Following the approach used by Marins [11] and attributed to [80], the covariance matrix for the quaternion components coming out of the Gauss-Newton routine, R_q, must be related to the known covariance matrix for the actual measurements, R_{sensor}. This relationship is developed [11] by first rewriting equation (9.16) as

$$\hat{q}_{k+1} = \hat{q}_k - F(\hat{q}_k)[\bar{y}_k^I - M(\hat{q}_k)\bar{y}_k^B]. \tag{9.21}$$

Next, multiplying through produces

$$\hat{q}_{k+1} = \hat{q}_k - F(\hat{q}_k)\bar{y}_k^I + F(\hat{q}_k)M(\hat{q}_k)\bar{y}_k^B. \tag{9.22}$$

As Marins points out, this shows that the quaternions coming out of the Gauss-Newton routine are a function of the previous estimate of the quaternion. Equation (9.22) can now be decomposed into a standard linear equation of the form

$$y = Ax + B \tag{9.23}$$

where

$$B = \hat{q}_k - F(\hat{q}_k)\bar{y}_k^I \tag{9.24}$$

and

$$A = F(\hat{q}_k)M(\hat{q}_k). \tag{9.25}$$

This matrix, A, is computed at each time step as the Gauss-Newton routine reaches convergence or exits due to reaching the maximum number of iterations. With the matrix A available, and having the known measurement covariance matrix for the measured vectors, the measurement covariance for the quaternion elements is found using [11]

$$R_q = AR_{sensor}A^T. \tag{9.26}$$

This "measurement" covariance matrix is then combined with the measurement covariance matrix of the rate sensors to form a 7×7 overall measurement covariance matrix.

For a real-world implementation, these sensor standard deviations must be determined and their squares are used to initialize the corresponding diagonal elements of R_{sensor} in the algorithm. In the case of the implementation using rates derived from differencing, finding the standard deviation of these "measurements" is problematic. For the cases used in this work, the actual standard deviation is calculated and then used. This information is not available in a real-world implementation and, currently, the only available solution is to simulation the motion ahead of time, and then use "typical values" for the matrix initialization. For the cases using actual rate sensors, this is not an issue. For the simulations run for this work, the standard deviations of all the simulated measurements are available and the squares of these values are used to initialize the R_{sensor} matrix.

9.7.2.4 Computing the State Transition Matrix, Φ

The State Transition Matrix, Φ, embodies the dynamic equations needed for the EKF to propagate the state vector forward in time. This is also the matrix that contains the linearization of the nonlinear dynamics in the form of a truncated Taylor Series approximation. The state space representation of the process model in Figure 9.3, assuming no inputs other than the white noise driving the rates, may be written as

$$\dot{\overline{x}} = F\overline{x} + \overline{w} \tag{9.27}$$

where F is the Dynamics Matrix representing the dynamic equations for the system, $\overline{x}$ is the state vector defined earlier, and $\overline{w}$ is a vector containing random, zero-mean, process noise. This process noise is meant to represent the known deficiencies in the dynamic model. Based on the process model, the dynamic equations for this system may be written as

$$\begin{aligned}
\dot{\omega}_1 &= \dot{x}_1 = \frac{-1}{\tau_{rx}} x_1 + \frac{1}{\tau_{rx}} w_{rx} \\
\dot{\omega}_2 &= \dot{x}_2 = \frac{-1}{\tau_{ry}} x_2 + \frac{1}{\tau_{ry}} w_{ry} \\
\dot{\omega}_3 &= \dot{x}_3 = \frac{-1}{\tau_{rz}} x_3 + \frac{1}{\tau_{rz}} w_{rz} \\
\dot{q}_1 &= \dot{x}_4 = \frac{1}{2\sqrt{x_4^2 + x_5^2 + x_6^2 + x_7^2}} (x_1 x_7 - x_2 x_6 + x_3 x_5) \\
\dot{q}_2 &= \dot{x}_5 = \frac{1}{2\sqrt{x_4^2 + x_5^2 + x_6^2 + x_7^2}} (x_1 x_6 + x_2 x_7 - x_3 x_4) \\
\dot{q}_3 &= \dot{x}_6 = \frac{1}{2\sqrt{x_4^2 + x_5^2 + x_6^2 + x_7^2}} (-x_1 x_5 + x_2 x_4 + x_3 x_7) \\
\dot{q}_4 &= \dot{x}_7 = \frac{1}{2\sqrt{x_4^2 + x_5^2 + x_6^2 + x_7^2}} (-x_1 x_4 - x_2 x_5 - x_3 x_6)
\end{aligned} \tag{9.28}$$

These are the equations, less the process noise terms in the first three, that are numerically integrated to propagate the states from one time step to the next. Clearly they are nonlinear in terms of the states and this is what drives the use of the EKF, and later the UKF approach. The EKF requires that this nonlinear system matrix be linearized. This is accomplished by taking partials of each of the dynamic equations with respect to each state and its time derivative according to

$$\dot{\bar{x}} = F\bar{x} \Rightarrow \Delta\dot{\bar{x}} = \left[\frac{\partial \dot{\bar{x}}}{\partial \bar{x}}\right]\Delta\bar{x} \tag{9.29}$$

where the linearized Dynamics Matrix, *F*, is defined as

$$F = \left.\frac{df(x)}{dx}\right|_{x=\hat{x}} . \tag{9.30}$$

Taking these partials yields the following non-zero entries for *F*

$$F(1,1) = \frac{\partial \dot{x}_1}{\partial x_1} = \frac{-1}{\tau_{rx}}$$

$$F(2,2) = \frac{\partial \dot{x}_2}{\partial x_2} = \frac{-1}{\tau_{ry}}$$

$$F(3,3) = \frac{\partial \dot{x}_3}{\partial x_3} = \frac{-1}{\tau_{rz}}$$

$$F(4,1) = \frac{\partial \dot{x}_4}{\partial x_1} = \frac{x_7}{2\sqrt{x_4^2 + x_5^2 + x_6^2 + x_7^2}}$$

$$F(4,2) = \frac{\partial \dot{x}_4}{\partial x_2} = \frac{-x_6}{2\sqrt{x_4^2 + x_5^2 + x_6^2 + x_7^2}}$$

$$F(4,3) = \frac{\partial \dot{x}_4}{\partial x_3} = \frac{x_5}{2\sqrt{x_4^2 + x_5^2 + x_6^2 + x_7^2}}$$

$$F(4,4) = \frac{\partial \dot{x}_4}{\partial x_4} = \left(x_1x_7 - x_2x_6 + x_3x_5\right)\frac{-x_4}{2\sqrt{x_4^2 + x_5^2 + x_6^2 + x_7^2}}$$

$$F(4,5) = \frac{\partial \dot{x}_4}{\partial x_5} = \frac{1}{2}\left\{\frac{x_3}{\sqrt{x_4^2 + x_5^2 + x_6^2 + x_7^2}} + \left(x_1x_7 - x_2x_6 + x_3x_5\right)\frac{-x_5}{\left(x_4^2 + x_5^2 + x_6^2 + x_7^2\right)^{\frac{3}{2}}}\right\}$$

$$F(4,6) = \frac{\partial \dot{x}_4}{\partial x_6} = \frac{1}{2}\left\{\frac{-x_2}{\sqrt{x_4^2 + x_5^2 + x_6^2 + x_7^2}} + \left(x_1x_7 - x_2x_6 + x_3x_5\right)\frac{-x_6}{\left(x_4^2 + x_5^2 + x_6^2 + x_7^2\right)^{\frac{3}{2}}}\right\}$$

$$F(4,7) = \frac{\partial \dot{x}_4}{\partial x_7} = \frac{1}{2}\left\{\frac{x_1}{\sqrt{x_4^2 + x_5^2 + x_6^2 + x_7^2}} + \left(x_1x_7 - x_2x_6 + x_3x_5\right)\frac{-x_7}{\left(x_4^2 + x_5^2 + x_6^2 + x_7^2\right)^{\frac{3}{2}}}\right\}$$

$$F(5,1)=\frac{\partial \dot{x}_5}{\partial x_1}=\frac{x_6}{2\sqrt{x_4^2+x_5^2+x_6^2+x_7^2}}$$

$$F(5,2)=\frac{\partial \dot{x}_5}{\partial x_2}=\frac{x_7}{2\sqrt{x_4^2+x_5^2+x_6^2+x_7^2}}$$

$$F(5,3)=\frac{\partial \dot{x}_5}{\partial x_3}=\frac{-x_4}{2\sqrt{x_4^2+x_5^2+x_6^2+x_7^2}}$$

$$F(5,4)=\frac{\partial \dot{x}_5}{\partial x_4}=\frac{1}{2}\left\{\frac{-x_3}{\sqrt{x_4^2+x_5^2+x_6^2+x_7^2}}+\left(x_1x_6+x_2x_7-x_3x_4\right)\frac{-x_4}{\left(x_4^2+x_5^2+x_6^2+x_7^2\right)^{\frac{3}{2}}}\right\}$$

$$F(5,5)=\frac{\partial \dot{x}_5}{\partial x_5}=\frac{1}{2}\left\{\left(x_1x_6+x_2x_7-x_3x_4\right)\frac{-x_5}{\left(x_4^2+x_5^2+x_6^2+x_7^2\right)^{\frac{3}{2}}}\right\}$$

$$F(5,6)=\frac{\partial \dot{x}_5}{\partial x_6}=\frac{1}{2}\left\{\frac{x_1}{\sqrt{x_4^2+x_5^2+x_6^2+x_7^2}}+\left(x_1x_6+x_2x_7-x_3x_4\right)\frac{-x_6}{\left(x_4^2+x_5^2+x_6^2+x_7^2\right)^{\frac{3}{2}}}\right\}$$

$$F(5,7)=\frac{\partial \dot{x}_5}{\partial x_7}=\frac{1}{2}\left\{\frac{x_2}{\sqrt{x_4^2+x_5^2+x_6^2+x_7^2}}+\left(x_1x_6+x_2x_7-x_3x_4\right)\frac{-x_7}{\left(x_4^2+x_5^2+x_6^2+x_7^2\right)^{\frac{3}{2}}}\right\}$$

$$F(6,1)=\frac{\partial \dot{x}_6}{\partial x_1}=\frac{-x_5}{2\sqrt{x_4^2+x_5^2+x_6^2+x_7^2}}$$

$$F(6,2)=\frac{\partial \dot{x}_6}{\partial x_2}=\frac{x_4}{2\sqrt{x_4^2+x_5^2+x_6^2+x_7^2}}$$

$$F(6,3)=\frac{\partial \dot{x}_6}{\partial x_3}=\frac{x_7}{2\sqrt{x_4^2+x_5^2+x_6^2+x_7^2}}$$

$$F(6,4)=\frac{\partial \dot{x}_6}{\partial x_4}=\frac{1}{2}\left\{\frac{x_2}{\sqrt{x_4^2+x_5^2+x_6^2+x_7^2}}+\left(-x_1x_5+x_2x_4+x_3x_7\right)\frac{-x_4}{\left(x_4^2+x_5^2+x_6^2+x_7^2\right)^{\frac{3}{2}}}\right\}$$

$$F(6,5)=\frac{\partial \dot{x}_6}{\partial x_5}=\frac{1}{2}\left\{\frac{-x_1}{\sqrt{x_4^2+x_5^2+x_6^2+x_7^2}}+\left(-x_1x_5+x_2x_4+x_3x_7\right)\frac{-x_5}{\left(x_4^2+x_5^2+x_6^2+x_7^2\right)^{\frac{3}{2}}}\right\}$$

$$F(6,6)=\frac{\partial \dot{x}_6}{\partial x_6}=\frac{1}{2}\left\{\left(-x_1x_5+x_2x_4+x_3x_7\right)\frac{-x_6}{\left(x_4^2+x_5^2+x_6^2+x_7^2\right)^{\frac{3}{2}}}\right\}$$

$$F(6,7)=\frac{\partial \dot{x}_6}{\partial x_7}=\frac{1}{2}\left\{\frac{x_3}{\sqrt{x_4^2+x_5^2+x_6^2+x_7^2}}+\left(-x_1x_5+x_2x_4+x_3x_7\right)\frac{-x_7}{\left(x_4^2+x_5^2+x_6^2+x_7^2\right)^{\frac{3}{2}}}\right\}$$

$$F(7,1)=\frac{\partial \dot{x}_7}{\partial x_1}=\frac{-x_4}{2\sqrt{x_4^2+x_5^2+x_6^2+x_7^2}}$$

$$F(7,2)=\frac{\partial \dot{x}_7}{\partial x_2}=\frac{-x_5}{2\sqrt{x_4^2+x_5^2+x_6^2+x_7^2}}$$

$$F(7,3)=\frac{\partial \dot{x}_7}{\partial x_3}=\frac{-x_6}{2\sqrt{x_4^2+x_5^2+x_6^2+x_7^2}}$$

$$F(7,4)=\frac{\partial \dot{x}_7}{\partial x_4}=\frac{1}{2}\left\{\frac{-x_1}{\sqrt{x_4^2+x_5^2+x_6^2+x_7^2}}+\left(-x_1x_4-x_2x_5-x_3x_6\right)\frac{-x_4}{\left(x_4^2+x_5^2+x_6^2+x_7^2\right)^{\frac{3}{2}}}\right\}$$

$$F(7,5)=\frac{\partial \dot{x}_7}{\partial x_5}=\frac{1}{2}\left\{\frac{-x_2}{\sqrt{x_4^2+x_5^2+x_6^2+x_7^2}}+\left(-x_1x_4-x_2x_5-x_3x_6\right)\frac{-x_5}{\left(x_4^2+x_5^2+x_6^2+x_7^2\right)^{\frac{3}{2}}}\right\}$$

$$F(7,6)=\frac{\partial \dot{x}_7}{\partial x_6}=\frac{1}{2}\left\{\frac{-x_3}{\sqrt{x_4^2+x_5^2+x_6^2+x_7^2}}+\left(-x_1x_4-x_2x_5-x_3x_6\right)\frac{-x_6}{\left(x_4^2+x_5^2+x_6^2+x_7^2\right)^{\frac{3}{2}}}\right\}$$

$$F(7,7)=\frac{\partial \dot{x}_7}{\partial x_7}=\frac{1}{2}\left\{\left(-x_1x_4-x_2x_5-x_3x_6\right)\frac{-x_7}{\left(x_4^2+x_5^2+x_6^2+x_7^2\right)^{\frac{3}{2}}}\right\}$$

(9.31)

The continuous time State Transition Matrix, Φ, is defined as

$$\Phi(t) = e^{Ft} \tag{9.32}$$

and the discrete form seen in (8.4) is generated by substituting the filter time step, T_s, for t. While e^{Ft} may be represented as an infinite Taylor Series expansion yielding

$$\Phi(t) = e^{Ft} = I + Ft + \frac{(Ft)^2}{2!} + \ldots + \frac{(Ft)^n}{n!} + \ldots, \tag{9.33}$$

this is not suitable for implementation. Unlike in a linear Kalman filter, where the state transition matrix is actually used to propagate the states forward one time step through matrix multiplication, in the EKF, this matrix is used only to facilitate the calculation of the Kalman gains. The states are actually propagated via numerical integration of the nonlinear dynamic equations. Given this usage, the infinite series is truncated to two terms, as additional terms do not generally improve performance [17]. This may be attributed to the fact that, for sufficiently small time steps, the first-order linearization is valid. Therefore, the discrete state transition matrix is approximated as

$$\Phi_k = e^{FT_s} \approx I + FT_s. \tag{9.34}$$

Substituting the values for F derived above and multiplying, the non-zero elements of Φ are

$$\Phi(1,1) = 1 - \frac{1}{\tau_{rx}} T_s$$

$$\Phi(2,2) = 1 - \frac{1}{\tau_{ry}} T_s$$

$$\Phi(3,3) = 1 - \frac{1}{\tau_{rz}} T_s$$

$$\Phi(4,4) = 1 + \frac{\partial \dot{x}_4}{\partial x_4} T_s$$

$$\Phi(4,5) = \frac{\partial \dot{x}_4}{\partial x_5} T_s$$

$$\Phi(4,6) = \frac{\partial \dot{x}_4}{\partial x_6} T_s$$

$$\Phi(4,7) = \frac{\partial \dot{x}_4}{\partial x_7} T_s$$

$$\Phi(5,1) = \frac{\partial \dot{x}_5}{\partial x_1} T_s$$

$$\Phi(5,2) = \frac{\partial \dot{x}_5}{\partial x_2} T_s$$

$$\Phi(5,3) = \frac{\partial \dot{x}_5}{\partial x_3} T_s$$

$$\Phi(5,4) = \frac{\partial \dot{x}_5}{\partial x_4} T_s$$

$$\Phi(5,5) = 1 + \frac{\partial \dot{x}_5}{\partial x_5} T_s$$

$$\Phi(5,6) = \frac{\partial \dot{x}_5}{\partial x_6} T_s$$

$$\Phi(5,7) = \frac{\partial \dot{x}_5}{\partial x_7} T_s$$

$$\Phi(6,1) = \frac{\partial \dot{x}_6}{\partial x_1} T_s$$

$$\Phi(6,2) = \frac{\partial \dot{x}_6}{\partial x_2} T_s$$

$$\Phi(6,3) = \frac{\partial \dot{x}_6}{\partial x_3} T_s$$

$$\Phi(6,4) = \frac{\partial \dot{x}_6}{\partial x_4} T_s$$

$$\Phi(6,5) = \frac{\partial \dot{x}_6}{\partial x_5} T_s$$

$$\Phi(6,6) = 1 + \frac{\partial \dot{x}_6}{\partial x_6} T_s$$

$$\Phi(6,7) = \frac{\partial \dot{x}_6}{\partial x_7} T_s$$

$$\Phi(7,1) = \frac{\partial \dot{x}_7}{\partial x_1} T_s$$

$$\Phi(7,2) = \frac{\partial \dot{x}_7}{\partial x_2} T_s$$

$$\begin{aligned}
\Phi(7,3) &= \frac{\partial \dot{x}_7}{\partial x_3} T_s \\
\Phi(7,4) &= \frac{\partial \dot{x}_7}{\partial x_4} T_s \\
\Phi(7,5) &= \frac{\partial \dot{x}_7}{\partial x_5} T_s \\
\Phi(7,6) &= \frac{\partial \dot{x}_7}{\partial x_6} T_s \\
\Phi(7,7) &= 1 + \frac{\partial \dot{x}_7}{\partial x_7} T_s
\end{aligned} \tag{9.35}$$

The values of the partials are those derived in equation (9.31) above.

9.7.2.5 Computing the Process Noise Matrix, *Q*

The Process Noise Matrix, *Q*, is meant to reflect the confidence in the overall system model. It is defined as

$$Q(t) = E[ww^T] \tag{9.36}$$

where *E* is the familiar expectation operator and *w* is the vector representing the process noise in the state-space system equation (8.2). The continuous time *Q(t)* is again a covariance matrix that is meant to reflect the uncertainty in the dynamic model of the system. The determination of this matrix, while seemingly straightforward, is probably the most difficult to accomplish. This stems from the difficulty in putting numbers to uncertainty in the model. It is initialized in this case as

$$Q = \begin{bmatrix} \Phi_s\sigma_{\omega_1}^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \Phi_s\sigma_{\omega_2}^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \Phi_s\sigma_{\omega_3}^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}. \tag{9.37}$$

Here, the sigmas are the estimated standard deviation of the rate measurements in each axis and Φ_s is a process noise tuning parameter added to account for uncertainty in the system model. Φ_s is unrelated to the state transition matrix Φ_k, and determination of the value for this tuning parameter is discussed in Chapter 10.. The discrete Q_k is calculated during each iteration of the EKF as

$$Q_k = \int_0^{T_s} \Phi(\tau) Q(t) \Phi^T(\tau) dt. \tag{9.38}$$

Substituting *Φ* from Section 9.7.2.5, integrating, and simplifying yields the following elements for Q_k

$$Q_k(1,1) = \Phi_s \left\{ \sigma_{\omega_1}^2 \left[T_s - \frac{T_s^2}{\tau_{rx}} + \frac{T_s^3}{3\tau_{rx}^2} \right] \right\}$$

$$Q_k(1,2) = Q_k(2,1) = 0$$

$$Q_k(1,3) = Q_k(3,1) = 0$$

$$Q_k(1,4) = \Phi_s \left\{ \sigma_{\omega_1}^2 \frac{\partial \dot{x}_4}{\partial x_1} \left[\frac{T_s^2}{2} - \frac{T_s^3}{3\tau_{rx}} \right] \right\}$$

$$Q_k(1,5) = \Phi_s \left\{ \sigma_{\omega_1}^2 \frac{\partial \dot{x}_5}{\partial x_1} \left[\frac{T_s^2}{2} - \frac{T_s^3}{3\tau_{rx}} \right] \right\}$$

$$Q_k(1,6) = \Phi_s \left\{ \sigma_{\omega_1}^2 \frac{\partial \dot{x}_6}{\partial x_1} \left[\frac{T_s^2}{2} - \frac{T_s^3}{3\tau_{rx}} \right] \right\}$$

$$Q_k(1,7) = \Phi_s \left\{ \sigma_{\omega_1}^2 \frac{\partial \dot{x}_7}{\partial x_1} \left[\frac{T_s^2}{2} - \frac{T_s^3}{3\tau_{rx}} \right] \right\}$$

$$Q_k(2,1) = Q_k(1,2) = 0$$

$$Q_k(2,2) = \Phi_s \left\{ \sigma_{\omega_2}^2 \left[T_s - \frac{T_s^2}{\tau_{ry}} + \frac{T_s^3}{3\tau_{ry}^2} \right] \right\}$$

$$Q_k(2,3) = Q_k(3,2) = 0$$

$$Q_k(2,4) = \Phi_s \left\{ \sigma_{\omega_2}^2 \frac{\partial \dot{x}_4}{\partial x_2} \left[\frac{T_s^2}{2} - \frac{T_s^3}{3\tau_{ry}} \right] \right\}$$

$$Q_k(2,5) = \Phi_s \left\{ \sigma_{\omega_2}^2 \frac{\partial \dot{x}_5}{\partial x_2} \left[\frac{T_s^2}{2} - \frac{T_s^3}{3\tau_{ry}} \right] \right\}$$

$$Q_k(2,6) = \Phi_s \left\{ \sigma_{\omega_2}^2 \frac{\partial \dot{x}_6}{\partial x_2} \left[\frac{T_s^2}{2} - \frac{T_s^3}{3\tau_{ry}} \right] \right\}$$

$$Q_k(2,7) = \Phi_s \left\{ \sigma_{\omega_2}^2 \frac{\partial \dot{x}_7}{\partial x_2} \left[\frac{T_s^2}{2} - \frac{T_s^3}{3\tau_{ry}} \right] \right\}$$

$$Q_k(3,1) = Q_k(1,3) = 0$$

$$Q_k(3,2) = Q_k(2,3) = 0$$

$$Q_k(3,3) = \Phi_s\left\{\sigma_{\omega_3}^2\left[T_s - \frac{T_s^2}{\tau_{rz}} + \frac{T_s^3}{3\tau_{rz}^2}\right]\right\}$$

$$Q_k(3,4) = \Phi_s\left\{\sigma_{\omega_3}^2\frac{\partial\dot{x}_4}{\partial x_3}\left[\frac{T_s^2}{2} - \frac{T_s^3}{3\tau_{rz}}\right]\right\}$$

$$Q_k(3,5) = \Phi_s\left\{\sigma_{\omega_3}^2\frac{\partial\dot{x}_5}{\partial x_3}\left[\frac{T_s^2}{2} - \frac{T_s^3}{3\tau_{rz}}\right]\right\}$$

$$Q_k(3,6) = \Phi_s\left\{\sigma_{\omega_3}^2\frac{\partial\dot{x}_6}{\partial x_3}\left[\frac{T_s^2}{2} - \frac{T_s^3}{3\tau_{rz}}\right]\right\}$$

$$Q_k(3,7) = \Phi_s\left\{\sigma_{\omega_3}^2\frac{\partial\dot{x}_7}{\partial x_3}\left[\frac{T_s^2}{2} - \frac{T_s^3}{3\tau_{rz}}\right]\right\}$$

$$Q_k(4,1) = Q_k(1,4)$$

$$Q_k(4,2) = Q_k(2,4)$$

$$Q_k(4,3) = Q_k(3,4)$$

$$Q_k(4,4) = \Phi_s\left\{\sigma_{\omega_1}^2\left(\frac{\partial\dot{x}_4}{\partial x_1}\right)^2 + \sigma_{\omega_2}^2\left(\frac{\partial\dot{x}_4}{\partial x_2}\right)^2 + \sigma_{\omega_3}^2\left(\frac{\partial\dot{x}_4}{\partial x_3}\right)^2\right\}\frac{T_s^3}{3}$$

$$Q_k(4,5) = \Phi_s\left\{\sigma_{\omega_1}^2\frac{\partial\dot{x}_4}{\partial x_1}\frac{\partial\dot{x}_5}{\partial x_1} + \sigma_{\omega_2}^2\frac{\partial\dot{x}_4}{\partial x_2}\frac{\partial\dot{x}_5}{\partial x_2} + \sigma_{\omega_3}^2\frac{\partial\dot{x}_4}{\partial x_3}\frac{\partial\dot{x}_5}{\partial x_3}\right\}\frac{T_s^3}{3}$$

$$Q_k(4,6) = \Phi_s\left\{\sigma_{\omega_1}^2\frac{\partial\dot{x}_4}{\partial x_1}\frac{\partial\dot{x}_6}{\partial x_1} + \sigma_{\omega_2}^2\frac{\partial\dot{x}_4}{\partial x_2}\frac{\partial\dot{x}_6}{\partial x_2} + \sigma_{\omega_3}^2\frac{\partial\dot{x}_4}{\partial x_3}\frac{\partial\dot{x}_6}{\partial x_3}\right\}\frac{T_s^3}{3}$$

$$Q_k(4,7) = \Phi_s\left\{\sigma_{\omega_1}^2\frac{\partial\dot{x}_4}{\partial x_1}\frac{\partial\dot{x}_7}{\partial x_1} + \sigma_{\omega_2}^2\frac{\partial\dot{x}_4}{\partial x_2}\frac{\partial\dot{x}_7}{\partial x_2} + \sigma_{\omega_3}^2\frac{\partial\dot{x}_4}{\partial x_3}\frac{\partial\dot{x}_7}{\partial x_3}\right\}\frac{T_s^3}{3}$$

$$Q_k(5,1) = Q_k(1,5)$$

$$Q_k(5,2) = Q_k(2,5)$$

$$Q_k(5,3) = Q_k(3,5)$$

$$Q_k(5,4) = Q_k(4,5)$$

$$Q_k(5,5) = \Phi_s\left\{\sigma_{\omega_1}^2\left(\frac{\partial\dot{x}_5}{\partial x_1}\right)^2 + \sigma_{\omega_2}^2\left(\frac{\partial\dot{x}_5}{\partial x_2}\right)^2 + \sigma_{\omega_3}^2\left(\frac{\partial\dot{x}_5}{\partial x_3}\right)^2\right\}\frac{T_s^3}{3}$$

$$Q_k(5,6) = \Phi_s \left\{ \sigma_{\omega_1}^2 \frac{\partial \dot{x}_5}{\partial x_1}\frac{\partial \dot{x}_6}{\partial x_1} + \sigma_{\omega_2}^2 \frac{\partial \dot{x}_5}{\partial x_2}\frac{\partial \dot{x}_6}{\partial x_2} + \sigma_{\omega_3}^2 \frac{\partial \dot{x}_5}{\partial x_3}\frac{\partial \dot{x}_6}{\partial x_3} \right\} \frac{T_s^3}{3}$$

$$Q_k(5,7) = \Phi_s \left\{ \sigma_{\omega_1}^2 \frac{\partial \dot{x}_5}{\partial x_1}\frac{\partial \dot{x}_7}{\partial x_1} + \sigma_{\omega_2}^2 \frac{\partial \dot{x}_5}{\partial x_2}\frac{\partial \dot{x}_7}{\partial x_2} + \sigma_{\omega_3}^2 \frac{\partial \dot{x}_5}{\partial x_3}\frac{\partial \dot{x}_7}{\partial x_3} \right\} \frac{T_s^3}{3}$$

$$Q_k(6,1) = Q_k(1,6)$$

$$Q_k(6,2) = Q_k(2,6)$$

$$Q_k(6,3) = Q_k(3,6)$$

$$Q_k(6,4) = Q_k(4,6)$$

$$Q_k(6,5) = Q_k(5,6)$$

$$Q_k(6,6) = \Phi_s \left\{ \sigma_{\omega_1}^2 \left(\frac{\partial \dot{x}_6}{\partial x_1}\right)^2 + \sigma_{\omega_2}^2 \left(\frac{\partial \dot{x}_6}{\partial x_2}\right)^2 + \sigma_{\omega_3}^2 \left(\frac{\partial \dot{x}_6}{\partial x_3}\right)^2 \right\} \frac{T_s^3}{3}$$

$$Q_k(6,7) = \Phi_s \left\{ \sigma_{\omega_1}^2 \frac{\partial \dot{x}_6}{\partial x_1}\frac{\partial \dot{x}_7}{\partial x_1} + \sigma_{\omega_2}^2 \frac{\partial \dot{x}_6}{\partial x_2}\frac{\partial \dot{x}_7}{\partial x_2} + \sigma_{\omega_3}^2 \frac{\partial \dot{x}_6}{\partial x_3}\frac{\partial \dot{x}_7}{\partial x_3} \right\} \frac{T_s^3}{3}$$

$$Q_k(7,1) = Q_k(1,7)$$

$$Q_k(7,2) = Q_k(2,7)$$

$$Q_k(7,3) = Q_k(3,7)$$

$$Q_k(7,4) = Q_k(4,7) \qquad (9.39)$$

$$Q_k(7,5) = Q_k(5,7)$$

$$Q_k(7,6) = Q_k(5,7)$$

$$Q_k(7,7) = \Phi_s \left\{ \sigma_{\omega_1}^2 \left(\frac{\partial \dot{x}_7}{\partial x_1}\right)^2 + \sigma_{\omega_2}^2 \left(\frac{\partial \dot{x}_7}{\partial x_2}\right)^2 + \sigma_{\omega_3}^2 \left(\frac{\partial \dot{x}_7}{\partial x_3}\right)^2 \right\} \frac{T_s^3}{3}$$

Again the values of the partials in these equations were determined in the derivation of F in equation (9.31).

9.7.2.6 Computing the Covariance Matrix Before the Update, *M*

The covariance matrix before the update, M, is calculated as

$$M = \Phi P \Phi^T + Q_k . \qquad (9.40)$$

All of these constituent matrices have been derived above with the exception of *P*, the covariance matrix after the update. While the process by which this matrix is initialized was addressed in Section 9.7.2.2, the derivation of the recurring matrix will be addressed in an upcoming section.

9.7.2.7 Propagating the State Vector Forward, $\bar{\mathbf{x}}$

In the case of a nonlinear system such as this, the state vector cannot be propagated forward in time using the state transition matrix as would be the case with a linear system. Instead, equations (9.28) are numerically integrated to propagate the states from one step to the next. In this implementation, a one-step Euler integration scheme is used.

9.7.2.8 Calculating the Projected Observations, $\bar{\mathbf{z}}$

In a general implementation of the EKF, the projected observations are a function, sometimes a nonlinear function, of the system states:

$$\bar{z} = h(\bar{x}) . \tag{9.41}$$

Because of the simplification gained by using the Gauss-Newton error minimization routine, in this implementation the projected observations are exactly equal to the projected states, or

$$\bar{z} = \bar{x} . \tag{9.42}$$

9.7.2.9 Calculating the Measurement Matrix, *H*

Typically, for a nonlinear system whose states are estimated by an EKF, the Measurement Matrix, *H*, is a linearization of the nonlinear function of the states, $h(\bar{x})$. Again, because the projected observations are simply the projected states, in this case the measurement matrix is

$$H = \left.\frac{dh(x)}{dx}\right|_{x=\bar{x}} = I_{7\times 7} \tag{9.43}$$

or simply equal to the identity matrix.

9.7.2.10 Computing the Kalman Gain Matrix, *K*

The Kalman gain is calculated at each time step as a function of the matrices we have already defined. The expression for this gain matrix is

$$K = MH^T[HMH^T + R]^{-1} . \tag{9.44}$$

9.7.2.11 Computing the Covariance Matrix After the Update, *P*

Once the Kalman gain matrix is available, the covariance matrix after the update, *P*, may be calculated. The expression used to do this is

$$P = (I - KH)M \ . \tag{9.45}$$

In essence, this provides a measure of how confident the filter is in its estimate at each step.

9.7.2.12 Computing the New State Estimate, $\hat{\mathbf{x}}$

Finally, the state estimate is created by the filter. Here, the filter combines the state vector that has been propagated forward by numerically integrating the nonlinear dynamic equations with a weighted residual formed by differencing the observation with the projected observation and weighting it with the Kalman gain. In this manner, the EKF produces its estimate of the state vector at the next time step. The equation used to calculate this estimate is

$$\hat{x} = \bar{x} + K(z^* - \bar{z}) \ . \tag{9.46}$$

where z^* is the measured observation from the sensors, or in this case from the rate sensors and from the Gauss-Newton routine in the case of the quaternion components.

Once this estimate is formed, the cycle begins again for the next time step as depicted in Figure 9.5. This entire process is implemented in the files norates_ekf.m and rates_ekf.m found in Appendix A.

9.8 Unscented Kalman Filter Design

The choice of the Unscented Kalman Filter (UKF) as the most promising estimation algorithm was detailed in Chapter 8. Figure 9.6 gives a pictorial view of the steps for implementing this routine. While the UKF is a fairly recent development, the general equations can be found in several good sources [67], [68], [69]. As an aid to the reader, the steps that differ from the traditional EKF are lightly shaded. The software implementation of the algorithm described here is found in Appendix A. The two files rates_ukf.m and norates_ukf.m implement the UKF routines for cases where rotational rate measurements come from sensors and where rotational rate "measurements" come from differencing, respectively. This section details the derivation of the various equations and matrices coded in these routines.

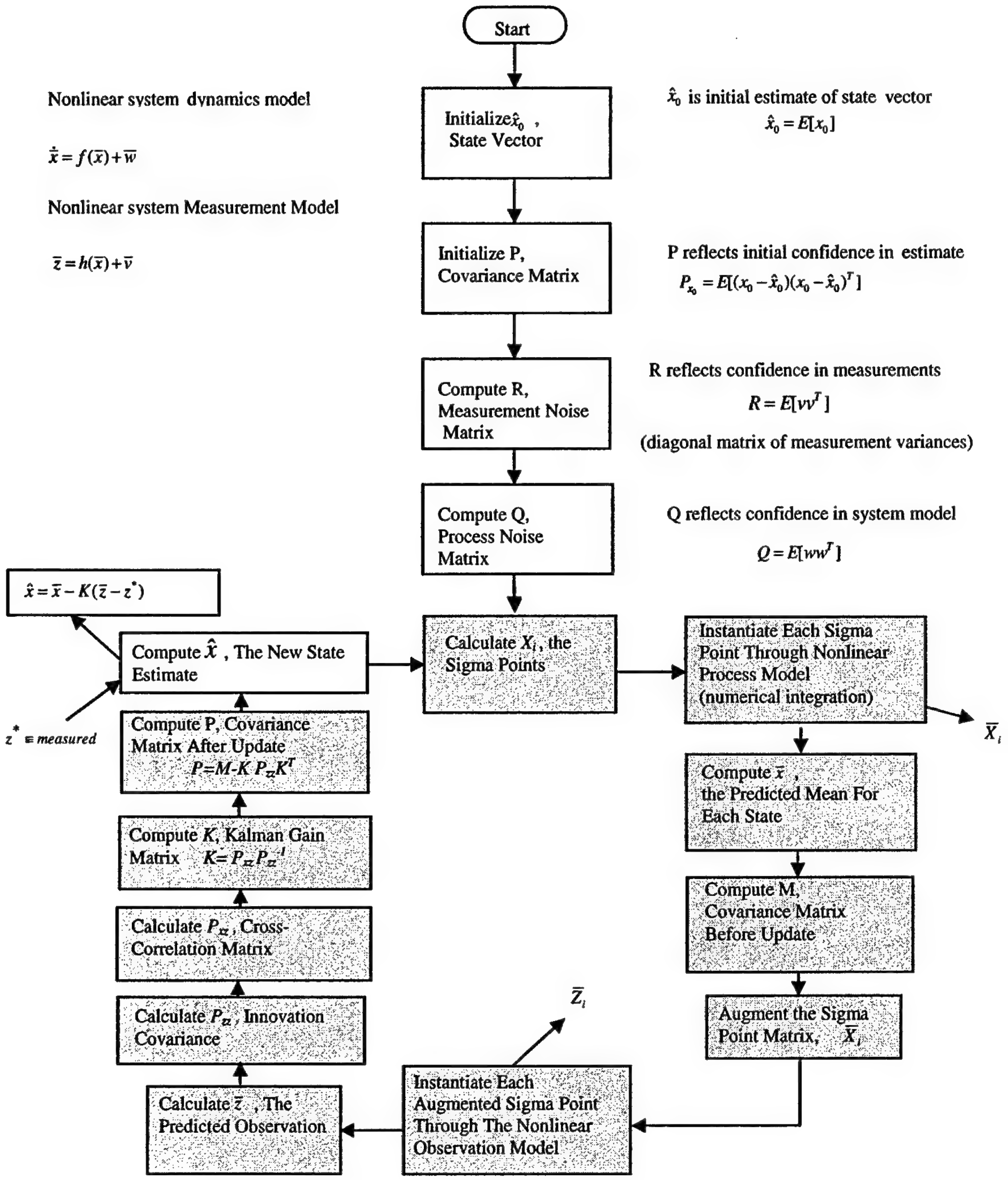

Start
Nonlinear system dynamics model
$\dot{\bar{x}} = f(\bar{x}) + \bar{w}$
Nonlinear system Measurement Model
$\bar{z} = h(\bar{x}) + \bar{v}$
Initialize $\hat{x}_0$, State Vector
$\hat{x}_0$ is initial estimate of state vector
$\hat{x}_0 = E[x_0]$
Initialize P, Covariance Matrix
P reflects initial confidence in estimate
$P_{x_0} = E[(x_0 - \hat{x}_0)(x_0 - \hat{x}_0)^T]$
Compute R, Measurement Noise Matrix
R reflects confidence in measurements
$R = E[vv^T]$
(diagonal matrix of measurement variances)
Compute Q, Process Noise Matrix
Q reflects confidence in system model
$Q = E[ww^T]$
$\hat{x} = \bar{x} - K(\bar{z} - z^*)$
Compute $\hat{x}$, The New State Estimate
$z^* \equiv measured$
Calculate X_i, the Sigma Points
Instantiate Each Sigma Point Through Nonlinear Process Model (numerical integration)
$\bar{X}_i$
Compute P, Covariance Matrix After Update $P = M - K P_{zz} K^T$
Compute $\bar{x}$, the Predicted Mean For Each State
Compute K, Kalman Gain Matrix $K = P_{xz} P_{zz}^{-1}$
Compute M, Covariance Matrix Before Update
Calculate P_{xz}, Cross-Correlation Matrix
Calculate P_{zz}, Innovation Covariance
$\bar{Z}_i$
Augment the Sigma Point Matrix, $\bar{X}_i$
Calculate $\bar{z}$, The Predicted Observation
Instantiate Each Augmented Sigma Point Through The Nonlinear Observation Model

Figure 9.6: Overall Block Diagram of Unscented Kalman Filter

9.8.1 Choice of State Vector

As with the EKF described in Section 9.7, the state vector to be estimated must describe the rotational motion of the vehicle. The UKF will be implemented to estimate the rotational rates that describe the dynamic motion and the quaternion elements that provide the attitude transformation at each time step. Therefore, the same state vector is used for both the EKF and the UKF and is repeated below.

$$\bar{x} = \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ q_1 \\ q_2 \\ q_3 \\ q_4 \end{bmatrix}. \tag{9.47}$$

9.8.2 Derivation and Definition of Filter Matrices

Many of the matrices derived for the UKF are similar, and sometimes are identical, to those used in the EKF-based algorithm. Since these were defined in detail in Section 9.7, only the differences will be elaborated on here.

9.8.2.1 Initializing the State Vector, $\hat{\mathbf{x}}_0$

The UKF filter routine also requires an initial guess, or estimate, of the state vector to begin processing. Realistically, at the initial time, a reasonable estimate of the rocket attitude and rates should be available. Since this work relies on a simulation environment, the user is prompted for the initial conditions, or the program defaults to the actual initial conditions with a user defined percent error. The initialization of the state vector in this work is always accomplished using the actual initial states plus a five percent error as described for the EKF in section 9.7. Also as with the EKF routine, if a filter "restart" was required during a real world flight, the snapshot provided by the Gauss-Newton routine could be used to reinitialize the filter.

9.8.2.2 Initializing the Covariance Matrix, *P*

The covariance matrix is a measure of the confidence in the estimate. As with the EKF, this matrix is defined as

$$P_{x_0} = E[(x_0 - \hat{x}_0)(x_0 - \hat{x}_0)^T] \tag{9.48}$$

where E symbolizes the "expectation" operator. For software implementation, this matrix is generated as in the same manner described in section 9.7, as a diagonal matrix with the squares of the error in the initial states along the diagonal.

9.8.2.3 Computing the Measurement Noise Matrix, *R*

The Measurement Noise Matrix, R, gives a reflection of the confidence in the accuracy of the measurements. It is defined as

$$R = E[vv^T] \tag{9.49}$$

where the symbol E is again the expectation operator and v is again the vector from (8.2) which represents the white noise process that corrupts the sensor measurements. Given the expectation operator in (9.49), this matrix may be interpreted as a covariance matrix of the measurement errors.

As described in 9.7.2.3, the Gauss-Newton routine produces a matrix, A, at each time step that relates the known measurement standard deviations to the derived quaternion component standard deviations.

$$R_q = AR_{sensor}A^T . \tag{9.50}$$

This 4×4 diagonal "measurement" covariance matrix is then combined with the measurement covariance matrix of the rate sensors to form a 7×7 diagonal overall measurement covariance matrix. For implementation purposes the sensor standard deviations are used to initialize the diagonal elements of R_{sensor}, subject to the same constraints described in Section 9.7.2.3.

9.8.2.4 Computing the Process Noise Matrix, *Q*

As is clear in Figure 9.6, the Process Noise Matrix, Q, does not change once it is initialized. For the UKF, the continuous time $Q(t)$, is defined as

$$Q(t) = E[ww^T] \tag{9.51}$$

where E is the familiar expectation operator and w is the error vector in the state-space system equation (8.2). Since the process noise matrix reflects the confidence in the system model, and lacking any significant rationale to increase one element over another, for the UKF implemented here the diagonal elements of $Q(t)$ are all initialized with the square of the user entered process noise tuning parameter, Φ_s.

$$Q = \begin{bmatrix} \Phi_s^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \Phi_s^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \Phi_s^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \Phi_s^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \Phi_s^2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \Phi_s^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \Phi_s^2 \end{bmatrix} \tag{9.52}$$

Although the process noise is often determined experimentally, as it will be here, Zarchan points out that "a good starting point is that the amount of process noise that a Kalman filter requires should reflect our lack of knowledge of the real world [17]." He elaborates that, "One method for determining process noise is to square the uncertainty in our expected initial error in the estimate and divide by the amount of filtering time." For this work, the optimum amount of process noise will be determined empirically. The actual "tuning" process for this parameter is described in Chapter 10.

9.8.2.5 Calculating the Sigma Points, χ_i

As described earlier, a number of "sigma points" must be calculated. These points essentially form a "cloud" of points that are then propagated forward as part of the estimation process. At each time step the sigma points, X_i, are calculated as [69],

$$\chi_{k-1} = \left\{ \hat{x}_{k-1} \quad \hat{x}_{k-1} + \gamma\sqrt{P_{k-1}} \quad \hat{x}_{k-1} - \sqrt{P_{k-1}} \right\} \tag{9.53}$$

for $k = 1,2,\ldots,\infty$ and γ is defined as

$$\gamma = \sqrt{L + \lambda} \tag{9.54}$$

where L is the dimension of the random variable, set equal to the dimension of the state vector to be estimated, or 7. Wan and van der Merwe [68] express the 2L+1 sigma points this way

$$\begin{aligned} \chi_0 &= \bar{x} \\ \chi_i &= \bar{x} + (\sqrt{(L+\lambda)P_x})_i, i = 1,\ldots,L \\ \chi_i &= \bar{x} - (\sqrt{(L+\lambda)P_x})_i, i = L+1,\ldots,2L \end{aligned} \tag{9.55}$$

where they note, for their implementation, that $(\sqrt{(L+\lambda)P_x})_i$ is the *i*th column of the matrix square root which in their approach is determined using a lower-triangular Cholesky factorization. In both cases, λ is a composite scaling factor defined as [68]

$$\lambda = \alpha^2 (L + \kappa) - L. \tag{9.56}$$

"The constant α determines the spread of the sigma points about $\bar{x}$ and is usually set to a small positive value (e.g., $10^{-4} \leq \alpha \leq 1$). The constant κ is a secondary scaling parameter which is usually set to 3-*L* … [68]." Further discussion of the selection of these parameters is found in Chapter 10.

9.8.2.6 Propagating the Sigma Points Forward in Time, $\bar{\chi}_i$

Once the "cloud" of sigma points is calculated, it must be propagated to the next time step. This is accomplished by "instantiating" each point through the nonlinear function. From an implementation standpoint, this means numerically integrating the nonlinear state dynamic equations forward one time step to get a transformed "cloud" of points at the next time step. The state dynamic equations are determined from the process model depicted in Figure 9.4 and are repeated here for continuity:

$$\begin{aligned}
\dot{\omega}_1 &= \dot{x}_1 = \frac{-1}{\tau_{rx}} x_1 + \frac{1}{\tau_{rx}} w_{rx} \\
\dot{\omega}_2 &= \dot{x}_2 = \frac{-1}{\tau_{ry}} x_2 + \frac{1}{\tau_{ry}} w_{ry} \\
\dot{\omega}_3 &= \dot{x}_3 = \frac{-1}{\tau_{rz}} x_3 + \frac{1}{\tau_{rz}} w_{rz} \\
\dot{q}_1 &= \dot{x}_4 = \frac{1}{2\sqrt{x_4^2 + x_5^2 + x_6^2 + x_7^2}} (x_1 x_7 - x_2 x_6 + x_3 x_5) \\
\dot{q}_2 &= \dot{x}_5 = \frac{1}{2\sqrt{x_4^2 + x_5^2 + x_6^2 + x_7^2}} (x_1 x_6 + x_2 x_7 - x_3 x_4) \\
\dot{q}_3 &= \dot{x}_6 = \frac{1}{2\sqrt{x_4^2 + x_5^2 + x_6^2 + x_7^2}} (-x_1 x_5 + x_2 x_4 + x_3 x_7) \\
\dot{q}_4 &= \dot{x}_7 = \frac{1}{2\sqrt{x_4^2 + x_5^2 + x_6^2 + x_7^2}} (-x_1 x_4 - x_2 x_5 - x_3 x_6).
\end{aligned} \tag{9.57}$$

In the case of the UKF implemented here, the instantiation of the sigma points is accomplished using a one-step Euler integration routine with a step size of .01 sec. The selection of this step size will be discussed in Chapter 10. This helps to minimize the time cost associated with propagating so many points through this numerical process.

9.8.2.7 Computing the Predicted Mean for Each State, $\bar{\mathbf{x}}$

Once the set of transformed sigma points is available, the mean state vector at the new time step may be calculated. This is accomplished by taking a weighted average of the transformed points, $\bar{\chi}_i$. The mathematical expression for this is

$$\bar{x} = \sum_{i=0}^{2L} W_i^{(m)} \bar{\chi}_i \tag{9.58}$$

where, again, the L is the dimension of the state vector as established earlier. The weights, W_i, are determined using [68]

$$\begin{aligned} W_0^{(m)} &= \frac{\lambda}{L+\lambda} \\ W_i^{(m)} &= \frac{1}{2(L+\lambda)}, i = 1,\ldots,2L \end{aligned} \tag{9.59}$$

where L and λ are as previously defined.

9.8.2.8 Computing the Covariance Matrix Before the Update, *M*

The covariance matrix before the update, M, is found using

$$M = \sum_{i=0}^{2L} \left\{ W_i^{(c)} (\bar{\chi}_i - \bar{x}) \right\} \left\{ \bar{\chi}_i - \bar{x} \right\}^T + Q \tag{9.60}$$

where L is again the dimension of the state vector, Q is the process noise matrix and $\bar{\chi}_i$ are the propagated sigma points. $\bar{\mathbf{x}}$ is the projected mean state vector. $W_i^{(c)}$ are again weights which are found using [68]

$$\begin{aligned} W_0^{(c)} &= \frac{\lambda}{L+\lambda} + 1 + \alpha^2 + \beta \\ W_i^{(c)} &= W_i^{(m)} = \frac{1}{2(L+\lambda)}, i = 1,\ldots,2L \end{aligned} \tag{9.61}$$

L, λ, and α are as previously defined and "β is used to incorporate prior knowledge of the distribution of **x** (for Gaussian distributions, β=2 is optimal) [68]."

9.8.2.9 Augmenting the Sigma Point Matrix, $\bar{\chi}_i$

As depicted in Figure 9.6, the next step in a UKF-based algorithm is to augment the set of projected sigma values to account for the process noise covariance, Q. There are two principle means to accomplish this

[68]. The first, used here, augments using additional points derived from the matrix square root of the process noise covariance and requires setting *L=2L* and recalculating the weights W_i. A second method augments by redrawing an entire new set of *L* points. While this results in fewer points, and therefore less propagation time, it also risks losing information that may have been captured by the original set of points that were then propagated forward. In this implementation, the first method is used for the sake of accuracy. If processing time becomes an issue in a real-time implementation, the second method should be further investigated.

The augmentation points added to the original set of transformed points are determined by

$$\begin{aligned} \chi_i &= \overline{\chi}_0 + (\sqrt{(L'+\lambda')Q})_i, i = L'+1,\ldots,\frac{3}{2}L' \\ \chi_i &= \overline{\chi}_0 - (\sqrt{(L'+\lambda')Q})_i, i = \frac{3}{2}L'+1,\ldots,2L' \end{aligned} \tag{9.62}$$

where *L'*=2*L*, and as before, $\sqrt{L'+\lambda'} = \gamma'$ and $\sqrt{Q}_i$ is the *i*th column of the matrix square root determined by the lower triangular Cholesky factorization. The parameter *λ'* must be recalculated based on the new values for *L'*, *α'*, and *κ'* due to the augmenting of the sigma points.

$$\lambda' = \alpha'^2 (L' + \kappa') - L' \tag{9.63}$$

α', and *κ'* are based on the same criteria used before augmenting the sigma point matrix. The weights are then recalculated based on the parameters for the augmented set according to

$$\begin{aligned} W_0^{(m)'} &= \frac{\lambda'}{L'+\lambda'} \\ W_i^{(m)'} &= \frac{1}{2(L'+\lambda')}, i = 1,\ldots,2L' \\ W_0^{(c)'} &= \frac{\lambda'}{L'+\lambda'} + 1 + \alpha'^2 + \beta' \\ W_i^{(c)'} &= W_i^{(m)'} = \frac{1}{2(L'+\lambda')}, i = 1,\ldots,2L' \end{aligned} \tag{9.64}$$

where *β'*=2 is optimal for a Gaussian distribution [68]. These new weights, based on the augmented sigma set, are then used for future calculations associated with the predicted observations and covariance matrices.

9.8.2.10 Propagating Prediction Points Forward in Time, $\overline{Z}_i$

Once the augmented set of sigma points, $\overline{\chi}_i'$, is available, each vector is "instantiated" through the observation model

$$\overline{Z}_i = h\left[\overline{\chi}_i'\right] \tag{9.65}$$

to get the predicted "observation cloud" based on the propagated sigma set. The function h is typically a nonlinear function of the state vector that has to be numerically propagated. Here, as with the EKF, the simplification due to using the Gauss-Newton routine becomes apparent. In this implementation, the predicted observations are exactly identical to the projected states. Therefore, in this case,

$$\overline{Z}_i = \overline{\chi}_i' \tag{9.66}$$

and no further propagation is necessary, thereby saving on computation time.

9.8.2.11 Computing the Predicted Observations, $\overline{\mathbf{z}}$

The predicted observations, $\overline{\mathbf{z}}$, are calculated as a weighted sum using the equation

$$\overline{z}_i = \sum_{i=0}^{2L'} W_i^{(m)'} \overline{Z}_i' = \sum_{i=0}^{2L'} W_i^{(m)'} \overline{\chi}_i' \, . \tag{9.67}$$

9.8.2.12 Calculating the Innovation Covariance, P_{zz}

The Innovation Covariance Matrix is calculated as follows

$$P_{zz} = \sum_{i=0}^{2L'} W_i^{(c)'} \left(\overline{Z}_i' - \overline{z}_i\right)\left(\overline{Z}_i' - \overline{z}_i\right)^T + R \, . \tag{9.68}$$

In this case, since $\overline{Z}_i = \overline{\chi}_i'$,

$$P_{zz} = \sum_{i=0}^{2L'} W_i^{(c)'} \left(\overline{\chi}_i' - \overline{z}_i\right)\left(\overline{\chi}_i' - \overline{z}_i\right)^T + R \tag{9.69}$$

where R is the measurement noise covariance matrix derived earlier.

9.8.2.13 Calculating the Cross Correlation Matrix, P_{xz}

The cross correlation matrix , P_{xz}, is also based on the "augmented weights" and is found using

$$P_{xz} = \sum_{i=0}^{2L'} W_i^{(c)'} \left(\overline{\chi}_i' - \bar{x}_i\right)\left(\overline{Z}_i' - \bar{z}_i\right)^T \tag{9.70}$$

or in this special case,

$$P_{xz} = \sum_{i=0}^{2L'} W_i^{(c)'} \left(\overline{\chi}_i' - \bar{z}_i\right)\left(\overline{\chi}_i' - \bar{z}_i\right)^T . \tag{9.71}$$

9.8.2.14 Computing the Kalman Gain Matrix, K

The Kalman gain matrix is computed using

$$K = P_{xz} P_{zz}^{-1} . \tag{9.72}$$

9.8.2.15 Computing the Covariance Matrix After the Update, P

P, the covariance matrix after the update, is a function of the Kalman gain, the covariance before the update, and the innovation covariance. The expression for calculating it is

$$P = M - KP_{zz}KT . \tag{9.73}$$

9.8.2.16 Computing the New State Estimate, $\hat{x}$

Finally, the goal is reached and the updated estimate of the state vector is computed using

$$\hat{x} = \bar{x} - K(\bar{z} - z^{*}) . \tag{9.74}$$

Here, $\bar{x}$ is the projected state, $\bar{z}$ is the projected observation, z^{*} is the "measurement" from the sensors, and K is the Kalman gain matrix.

10.0 Tuning of the Algorithm and Validation of Performance

With a firm grasp of the design of the EKF and UKF algorithms in hand, it is now time to describe the data each will operate on and how each of the parameters required to execute the algorithms is determined. This chapter addresses both these topics as well as providing examples of each algorithm's performance.

10.1 Computer Simulation of True Motion

Due to a paucity of real, high fidelity attitude data for a rocket, the data for development and test of these algorithms is simulated. Both the simulation and the algorithms are programmed using the software package MATLAB. The advantage of using simulation data is that the absolute true values are known. These values are corrupted to provide simulated measurements, and then filtered in hopes of returning to the true data, which is available for comparison. When using real data, the truth is seldom known, and at best the experimenter is left to compare with data that, hopefully, is measured by a system with equal or better accuracy than that being tested. The disadvantage, of course, is that the simulation may not adequately reflect the real world. The simulation tool developed here allows the user to generate certain body rate profiles, which are then translated into rocket attitude motion. Other, more complex, profiles can also be developed off-line and, with minor manipulation of the script attitude_sim.m, input to the overall algorithm for testing. The profiles included are intended to simulate motion representative of that expected in a sounding rocket. They are not intended to provide a high fidelity simulation of exact rocket motion, which is extremely complex. This approach assumes that if the attitude estimation algorithm can successfully process the motions simulated here, it should be able to successfully estimate attitude from real sensor data. The next sections describe the methodology used to produce simulated data for algorithm development and testing.

10.1.1 Methodology for Simulating Motion

The following diagram depicts the process by which the true motion, and measurement of that motion, is simulated. The user-interactive code to accomplish the steps shown below is in the MATLAB file attitude_sim.m and is found in Appendix A. The desired rotational rate profile indicated in the "initialization" block in Figure 10.1 is chosen by the user while running attitude_sim.m. The choices consist of fixed rates about each axis, a linearly ramped profile, an exponential profile that asymptotically approaches a final value, or a step/pulse profile. The profiles are implemented in the programs omegagen_fix.m, omegagen_ramp.m, omegagen_exp.m and omegagen_step.m respectively.

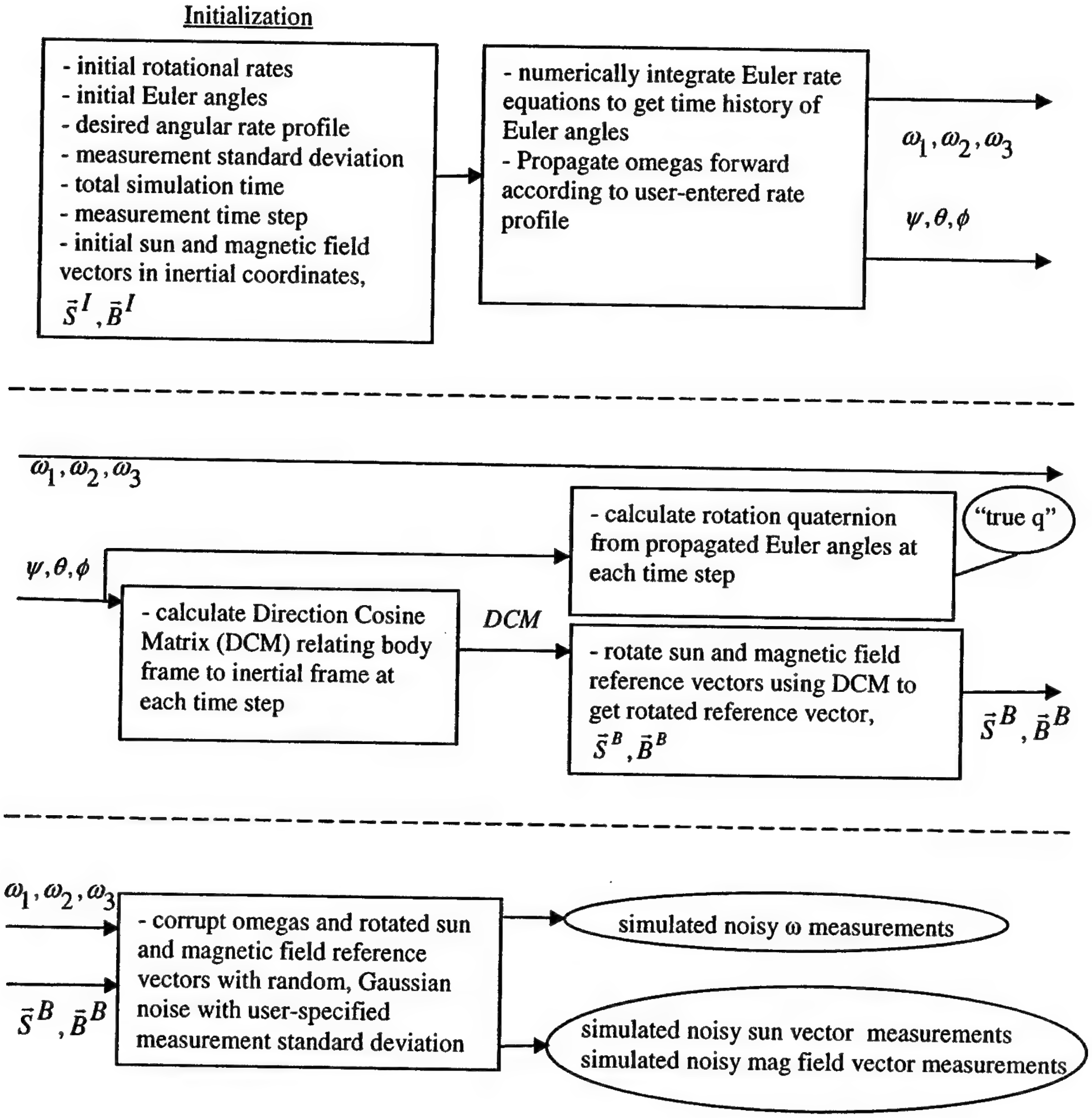

Initialization
- initial rotational rates
- initial Euler angles
- desired angular rate profile
- measurement standard deviation
- total simulation time
- measurement time step
- initial sun and magnetic field vectors in inertial coordinates, $\vec{S}^I, \vec{B}^I$
- numerically integrate Euler rate equations to get time history of Euler angles
- Propagate omegas forward according to user-entered rate profile
$\omega_1, \omega_2, \omega_3$
ψ, θ, ϕ
$\omega_1, \omega_2, \omega_3$
ψ, θ, ϕ
- calculate rotation quaternion from propagated Euler angles at each time step
"true q"
- calculate Direction Cosine Matrix (DCM) relating body frame to inertial frame at each time step
DCM
- rotate sun and magnetic field reference vectors using DCM to get rotated reference vector, $\vec{S}^B, \vec{B}^B$
$\vec{S}^B, \vec{B}^B$
$\omega_1, \omega_2, \omega_3$
$\vec{S}^B, \vec{B}^B$
- corrupt omegas and rotated sun and magnetic field reference vectors with random, Gaussian noise with user-specified measurement standard deviation
simulated noisy ω measurements
simulated noisy sun vector measurements
simulated noisy mag field vector measurements

Figure 10.1: Generation of Simulated Data and Measurements

These are called by attitude_sim.m based on user input, and the code itself may be found in Appendix A. In each case, the user is able to define a number of parameters, and the software generates body rates at each time step for use by attitude_sim.m.

From the diagram, it may be seen that the "true" motion is propagated forward by a numerical integration of the Euler equations, making use of the user defined rate profile. In this implementation a second-order Runge-Kutta integration scheme is used to integrate equations (5.3). This propagation of the motion could also be done by integrating a quaternion rate profile, using equation (4.22). However, since the goal is to generate estimates of the attitude quaternion, and evaluate the results of an attitude quaternion filter, propagating the Euler angles and then converting the propagated value to a quaternion provides some separation in methods. This in turn gives more confidence that the filter is truly estimating the quaternion from the measurements and not somehow reiterating what it has been given. This same approach was used in [11]. Of course, a complication associated with the use of Euler angles is the occurrence of singularities at certain orientations. While the code includes protections against "divide by zero," etc., the user should be aware of the possibility of these singularities and investigate any aberrations in the simulated motion with these in mind.

Once the Euler angles are propagated to the next time step, equation (4.27) is used to calculate the "true" propagated quaternion corresponding to these angles. The propagated Euler angles are also used to calculate a Direction Cosine Matrix (DCM) using equation (4.1). This DCM is used to rotate the reference vector defined in the inertial frame to one defined in the body frame, producing what a perfect sensor suite should measure. Next, both the rotated reference vector and the rotational rates for a given time step have random noise added to simulate noisy measurements at that time step. The random noise is Gaussian with a standard deviation equal to that defined by the user.

To summarize, attitude_sim.m produces arrays holding the simulated "true" quaternion, noisy Sun sensor and magnetometer measurements, and noisy rotational rate measurements for each time step. These array elements are then read one at a time by the attitude determination algorithm to simulate the sequential availability of the data.

10.1.2 Single 90 Degree Rotation

To demonstrate the correct function of the data simulation implemented in attitude_sim.m, several simple cases are run. These are cases which can be easily visualized, yet some present situations which would cause difficulties due to singularities. The first is a single rotation of 90 degrees about one of the principle axes. While rotations about each axis were investigated, a positive rotation about the "z" axis is discussed in this section. This rotation is easily visualized, and corresponds to an Euler angle sequence of $\psi=90°$, $\theta=0°$, $\varphi=0°$, or equivalently, $\psi=0°$, $\theta=0°$, $\varphi=90°$. Following the rotation, the rotated "x" axis points in the direction of the unrotated "y" axis and the rotated "y" axis points in the direction of the original "-x" axis.

The "z" axis remains unchanged since this is a single rotation about the "z" axis. The results of simulating this motion using attitude_sim.m are shown in Figure 10.2 below. Keep in mind that the body frame is rotating according to the Euler sequence described above. As a result, the (1, 0, 0) inertial vector, depicted here as the Sun vector, appears in the rotated frame to have the coordinates (0, -1, 0), as expected. Also as expected, the (0, 0, 1) inertial vector, depicted here as the magnetic field vector, has unchanged coordinates in the rotated frame.

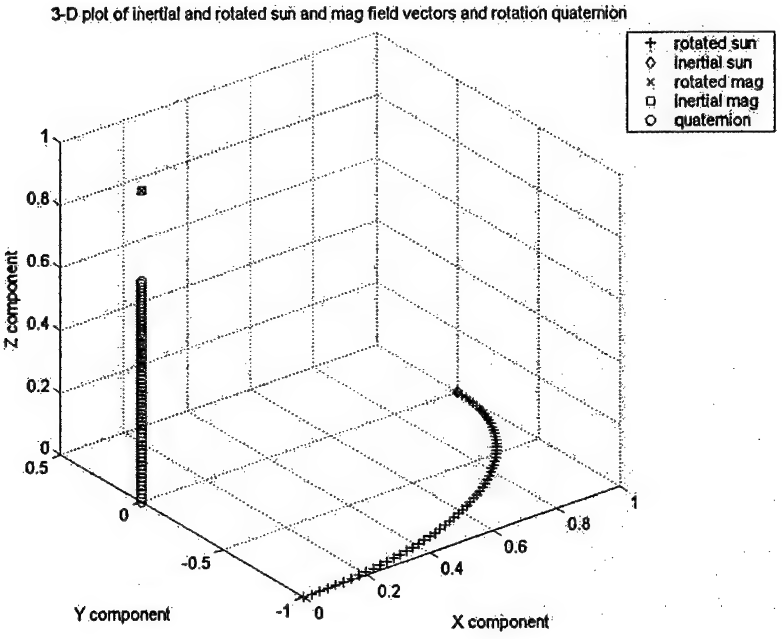


Figure 10.2: Single 90 Degree Rotation as Depicted by Attitude_sim.m

Using equation (4.13), the corresponding attitude quaternion is q = (0, 0, 0.7071, 0.7071). Figure 10.3 below demonstrates that attitude_sim.m produces this expected result. Finally, as described above, this rotation corresponds to the Euler angle sequence $\psi=0^\circ$, $\theta=0^\circ$, $\varphi=90^\circ$. As seen in Figure 10.4, attitude_sim.m produces these results as well. The single 90 degree rotation performance is similar for rotations about either of the other two axes.

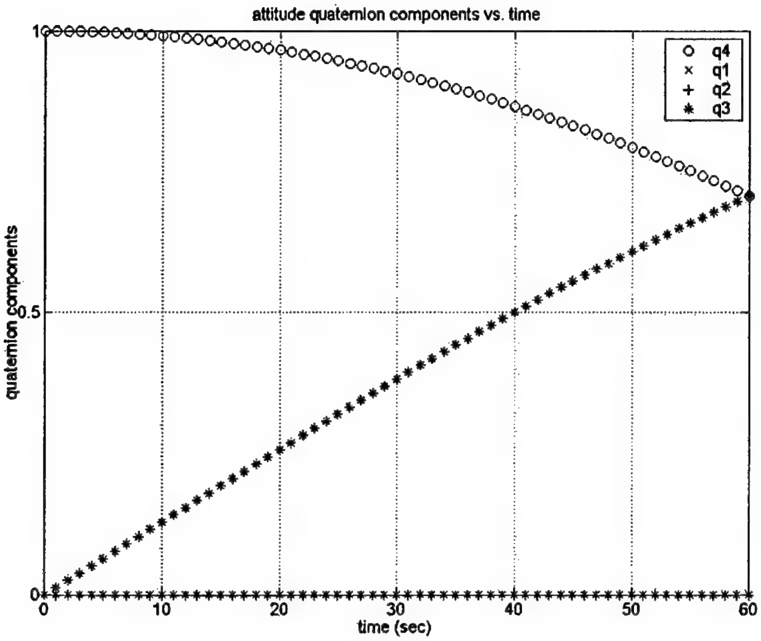


Figure 10.3: Quaternion Elements for Single 90 Degree Rotation as Produced by Attitude_sim.m

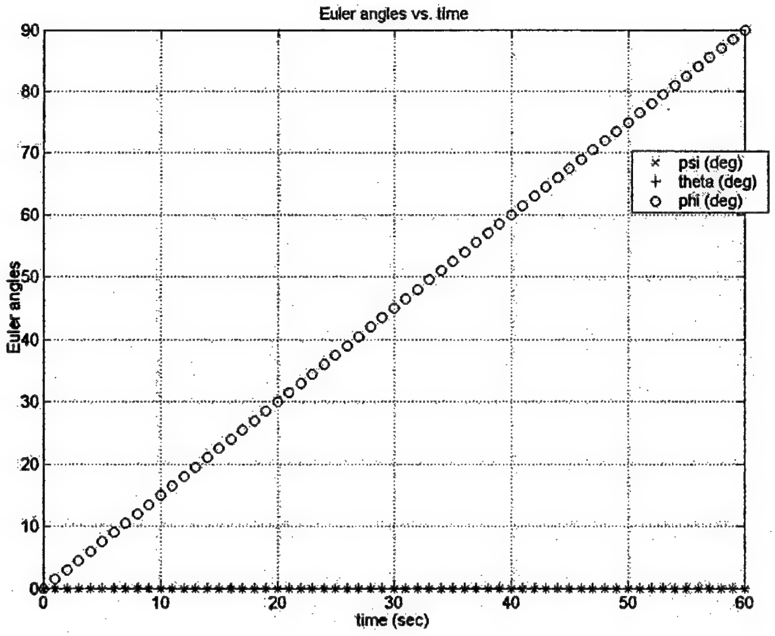


Figure 10.4: Euler Angles for Single 90 Degree Rotation as Produced by Attitude_sim.m

10.1.3 Two Successive 90 Degree Rotations

The last section demonstrated the results of the easily visualized single 90 degree rotation about the third axis. In this section, similar results are presented for the case of a 90 degree rotation about the "y" axis followed by a 90 degree rotation about the "z" axis. Figure 10.5 below illustrates the inertial (1, 0, 0) "Sun" vector as seen from the rotated frame. Its coordinates are (0, 0, 1) as expected. Similarly, the inertial (0, 0, 1) "mag field" vector is represented as (0, 1, 0) in the rotated frame. This also is as expected. In the figures in this section, the motion is simulated at a .01 sec time step, but only every 100th point is plotted for clarity.

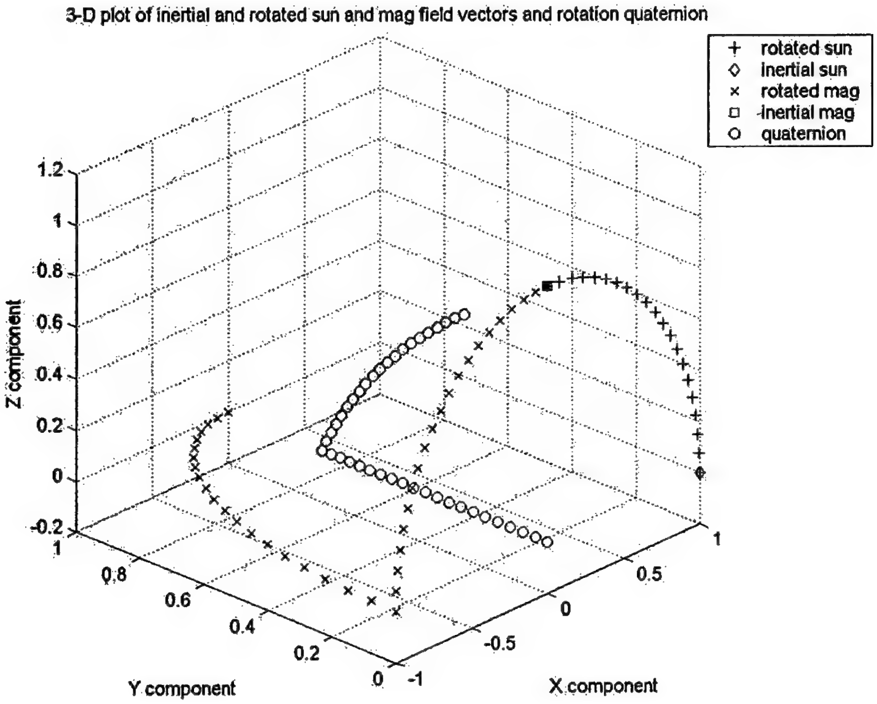


Figure 10.5: Successive 90 Degree Rotations as Depicted by Attitude_sim.m

Again, based on equation (4.13), the rotation quaternion for the first rotation is (0, .7071, 0, .7071) and for the second rotation it is (0, 0, .7071, .7071). Using equation (4.21) to multiply these two quaternions, we see that the combined rotation is represented by the quaternion (.5, .5, .5, .5). Figure 10.6, below,

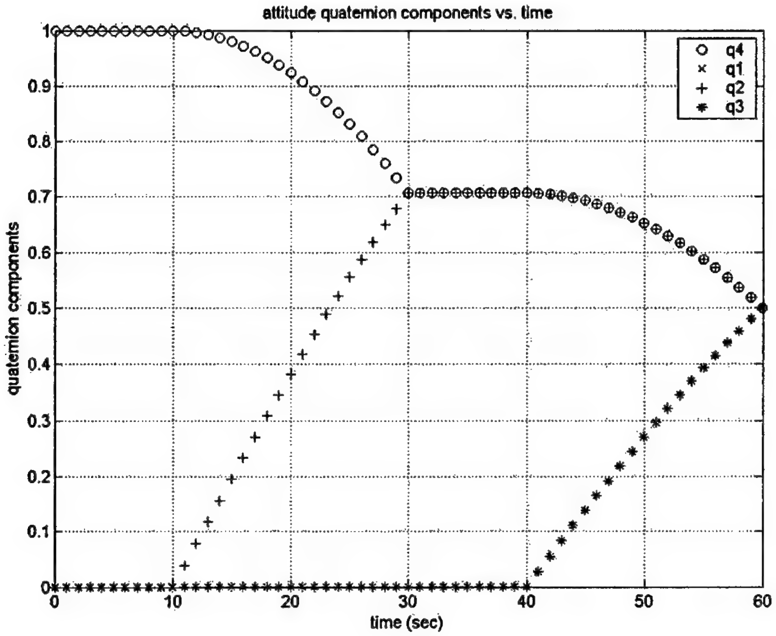


Figure 10.6: Quaternion Elements for Successive 90 Degree Rotations as Produced by Attitude_sim.m

illustrates that these results are obtained from attitude_sim.m. The first rotation takes the body frame from the Euler angles $\psi=0^\circ$, $\theta=0^\circ$, $\varphi=0^\circ$, to $\psi=90^\circ$, $\theta=90^\circ$, $\varphi=270^\circ$,. The second takes the frame to a final attitude of $\psi=90^\circ$, $\theta=90^\circ$, $\varphi=0^\circ$. Figure 10.7 illustrates the successful generation of these angles by attitude_sim.m. This set of rotations is a good test of the simulation software since it encompasses orientations that could cause difficulty due to the singularities inherent in the Euler angle equations. As illustrated here, the expected results are achieved despite this.

10.1.4 Single 120 Degree Rotation

As a final demonstration of the ability of attitude_sim.m to correctly simulate desired rotations, this final section gives the results for a single rotation about a vector in the direction of (1,1,1) with a magnitude of 120 degrees. This is another easily visualized rotation that allows for validation of correct simulation. The results of this rotation should match those from the last section since a single rotation of 120 degrees about (1, 1, 1) is equivalent to successive 90 degree rotations about the second axis and then about the third axis. The simulation is generated with a constant rotational rate of 0.19245 rev/min about each body axis for a duration of 60 sec. This rate is determined by calculating the equal rate about each axis that will produce a rotation of 1/3 revolution, or 120 degrees. Figure 10.8 below gives the overall view of the motion. Again,

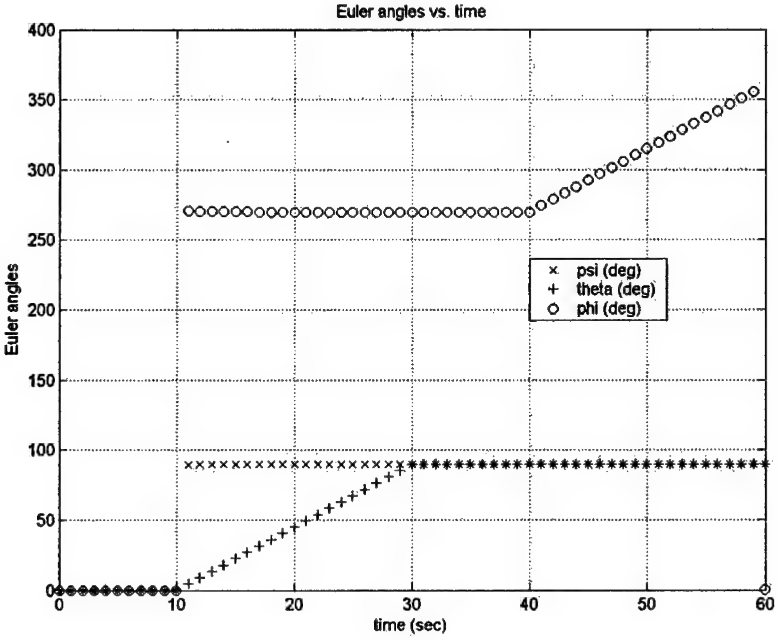


Figure 10.7: Euler Angles for Successive 90 Degree Rotations as Produced by Attitude_sim.m

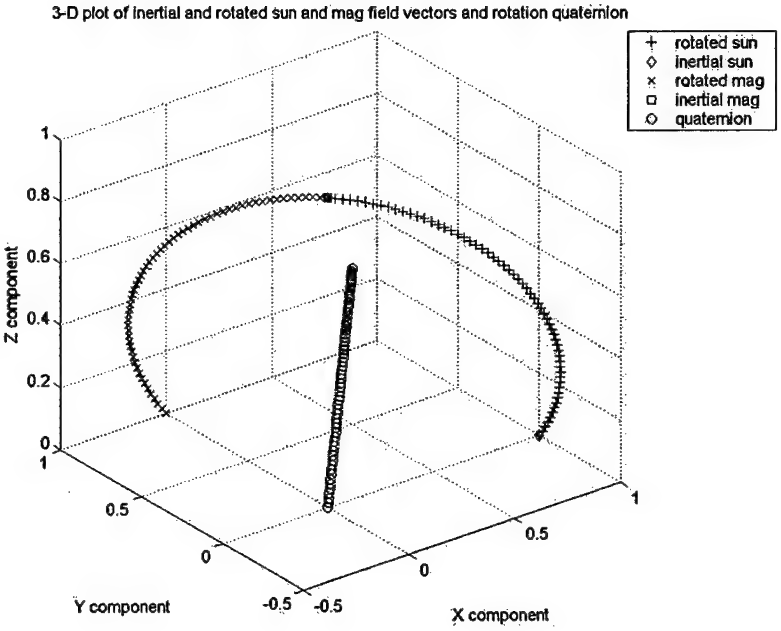


Figure 10.8: Single 120 Degree Rotation About (1,1,1)

the inertial (1, 0, 0) "Sun" vector has coordinates (0, 0, 1) as seen from the rotated frame. The inertial (0, 0, 1) "mag field" vector is represented as (0, 1, 0) in the rotated frame. These results do indeed match those of the last section, and those that can be visualized. For these figures, the motion is again simulated at a .01 sec time step, but only every 100th point is plotted for clarity. The next figure demonstrates the quaternion elements, which again match those of the previous section, as expected. Using equation (4.13), it is clear that (.5, .5, .5, .5) is the correct quaternion for a 120 degree rotation about an axis in the direction of (1, 1, 1). Finally, Figure 10.10 depicts the Euler angles for this rotation. The set ψ=90°, θ=90°, φ=0° matches the expected result.

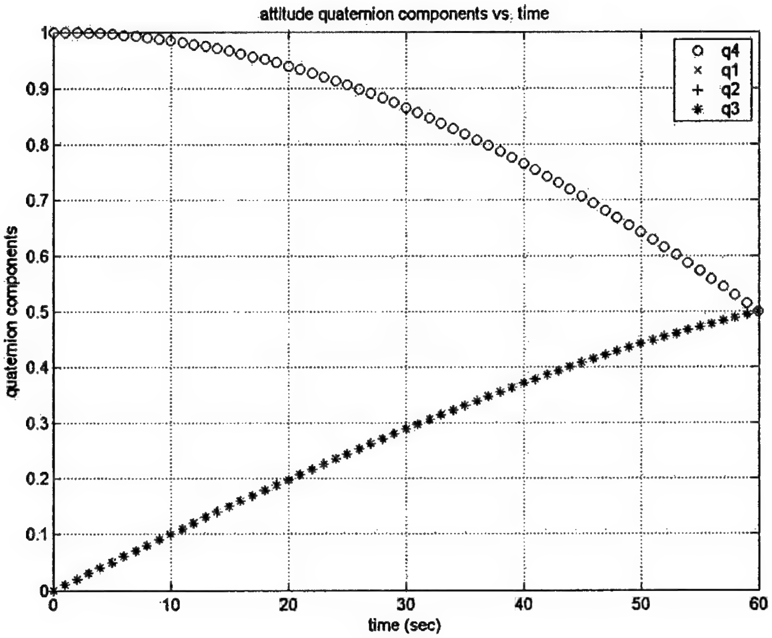


Figure 10.9: Quaternion Elements for Single 120 Degree Rotation About (1,1,1)

These results demonstrate that attitude_sim.m generates the desired results given a rotational rate profile. In each case, the initial Euler angles are ψ=0°, θ=0°, φ=0°, corresponding to the body frame aligned with the inertial frame. The rate profile then causes a rotation of the body frame to a final orientation. While these three cases are a small subset of those simulated, they represent easily visualized situations where validation of the results is facilitated. At the same time, they encompass orientations that pose potential difficulties due to singularities. In each case, the expected results are faithfully produced.

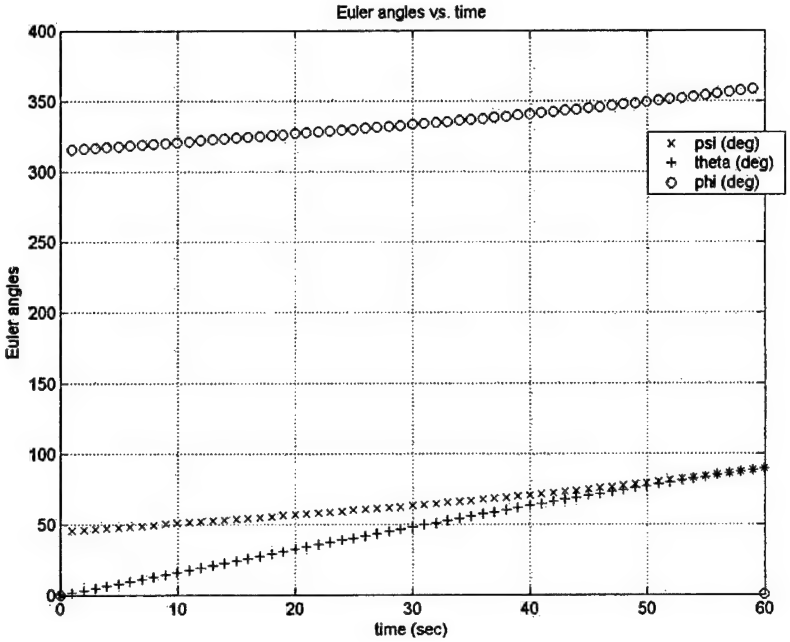


Figure 10.10: Euler Angles for Single 120 Degree Rotation About (1,1,1)

10.2 Computer Simulation of Noisy Sensor Measurements

In addition to simulating the "true" motion of the object, attitude_sim.m simulates noisy measurements. This section discusses how this is accomplished.

10.2.1 Corrupting "Truth" with Noise to Simulate Sensor Measurements

Attitude_sim.m has the capability to simulate noisy Sun sensor measurements, noisy magnetic field sensor measurements, and noisy rate measurements. For the two vectors, random noise having a user-defined standard deviation is added to each component at each time step. This noise is zero-mean Gaussian noise, and a different random value is added to each component. Similarly, zero-mean Gaussian noise is added to each rotational rate at each time step. In this manner, sensor measurements are simulated that have random noise with a standard deviation equal to the expected sensor accuracy. While real world noise is not often truly white, zero-mean, and Gaussian, it is often simulated this way for a number of reasons. First, lacking a better criterion for what noise to expect, Gaussian noise is an oft-made assumption that produces good results. Second, as Zarchan points out, while "white noise is not physically realizable, it can be used to serve as an invaluable approximation to situations in which a disturbing noise is wideband compared to the system bandwidth. In addition, white noise is useful for analytical operations because of the impulsive nature of the autocorrelation function (it makes integrals disappear) [16]." In this case, where the data is being simulated and where no better model for the noise exists, Gaussian noise is assumed for the reasons stated above. The software file, attitude_sim.m, could be modified relatively easily for future simulations where a different noise profile is desired.

10.2.2 Simulating a Sensor or Measurement Bias

In order to simulate a bias on one or more sensors, a set bias may be added to each array value generated by attitude_sim.m. While this capability is not automated in the current code, this operation is easily accomplished in MATLAB. In this manner, a separate bias may be added to each component of each sensor, providing great flexibility in evaluating performance.

10.2.3 Simulating Data "Dropouts"

Similar to the situation with adding bias to measurements, a drop-out may be simulated in any, or all, of the sensor measurements. The capability to simulate dropouts of all Sun sensor measurements for a period, all magnetometer measurements for a period, or all rate data for a period is currently built into attitude_sim.m as a user selected option. If implemented, the user enters which sensor to drop out, the number of measurements to be lost, and at what point in the simulation the drop-out should begin. These drop-outs consist of the sensor data being set to zero for these measurements. In this fashion, the user can analyze the effect of a loss of signal or a failed or intermittent sensor.

10.3 Measurement of Algorithm Performance

In order to judge how well the algorithm is estimating attitude motion, some measure of performance must be defined. For this effort, measures of performance are based on mean-square-error between the "true" values and those estimated by the algorithm. This section details how the measures are defined for each quantity estimated.

10.3.1 Attitude Estimation Performance

The attitude estimate from the algorithm is in the form of an attitude quaternion. Since the true quaternion is available from attitude_sim.m, a straight comparison of quaternion components could be accomplished to judge performance. However, since the quaternion is essentially describing a transformation between the body frame and the inertial frame, a more telling measure of performance is a comparison of the known reference vector rotated by the "true" quaternion and that rotated by the estimated quaternion. Therefore, at each time step, the reference vector composed of the inertial Sun vector concatenated with the inertial magnetic field vector is rotated using the "true" attitude quaternion for that time step provided by attitude_sim.m. Also at each time step, the best estimate of the quaternion from the algorithm is used to rotate the inertial reference vector to the body frame. Then the mean square error is calculated between the two rotated vectors. This value is used to measure how well a particular method using a given set of parameters performs in terms of estimating attitude at each time step.

10.3.2 Rotational Rate Estimation Performance

The algorithm also estimates rotational rates. The accuracy of this estimate must also be assessed. Again, a mean square error criterion is used. Here the "true" rotational rates at each time step are known from the user defined rate profile. The mean square error of the estimated rate vector is determined at each time step by comparing the rates about each of the body axes. This value is used to measure how well the algorithm is estimating the overall rate vector at each time step.

10.3.3 Overall Figure of Merit

Rather than attempt to scrutinize the attitude and rate performance at each time step, an overall figure of merit is defined to judge performance over the length of an entire simulation run. This aids in comparing the results from various algorithms using various parameters without a painstaking time step by time step analysis. First, the overall attitude estimation performance is calculated by taking an average of the mean square error over all the time steps in a run. Second, the overall rate estimation performance is calculated by taking an average of the mean square error over all the time steps in the run. Finally, these two are combined to yield the overall "figure of merit." The two are combined by taking the square root of the sum of the squares of the two averages. This provides a single number to judge comparative performance between different algorithms, or between separate filtering runs on the same data, by the same algorithm, but with different parameters. Certainly, the attitude or rate performance can still be evaluated independently since the average of the mean square error is available for each.

10.4 Error Minimization via Gauss-Newton

The software to implement the Gauss-Newton error minimization routine is found in Appendix A. As developed in Chapter 9, this part of the estimation algorithm has a number of parameters that must be defined. From a programming standpoint, it is necessary to determine what the convergence criteria will be and what maximum number of iterations will be allowed. As part of this effort, a study of the number of iterations versus error performance was conducted. As reported by Marins [11], the algorithm nearly always converges in very few steps, often in one or two. The application in that work always began with a first guess very near the true value. In this application, the guess will be relatively close to the true value, but due to the nature of the low-cost sensors, is often further from the true value than was encountered by Marins. As a result, it is important to evaluate what the optimum parameters are and to demonstrate the convergence properties of the Gauss-Newton routine.

10.4.1 Convergence Tolerance and the Optimum Number of Steps

A large number of runs were conducted using the gauss_newton.m routine to examine its performance. As developed in Chapter 9, the routine determines the "best" rotation quaternion that minimizes the error between the known true reference vector and the measured vector. With an emphasis on minimizing run time, in order to allow real-time implementation, there may be a maximum number of iterations that should be allowed as the routine attempts to minimize error. In other words, even if the convergence criterion, or tolerance, has not been met, the routine will stop iterating and produce an estimate of the rotation quaternion. In this implementation, the convergence criteria, or tolerance, is the mean square error found using the ε error vector defined earlier. Since the sensors provide noisy measurements, in some few instances, the minimum error achieved does not always converge to a value that meets the convergence criteria. In some situations, the minimization produces a steady-state error that allows for a useful estimated quaternion to be produced, but that will not shrink appreciably no matter how many iterations are allowed. In this case, in order to minimize run time, it is beneficial to terminate the loop at some small number of iterations. The question becomes one of how many iterations should be allowed. Since the routine does an excellent job of minimizing the error in a small number of steps, this maximum number may be set at a low value. This holds true even with the noisy sensors employed here. The following table shows the results from a sample of simulation runs conducted and the resulting overall attitude performance for various combinations of convergence tolerance and maximum number of iterations. From this limited excerpt of the results, it may be seen that the best overall performance, in terms of minimum error while minimizing the run time, is achieved using a tolerance of 0.01 and a maximum number of iterations set to 10. These are the parameters employed in the Gauss-Newton portion of the algorithm.

Table 10.1: Gauss-Newton Performance for Given Convergence Tolerance and Maximum Iterations

		20 sec run	20 sec run	20 sec run	120 sec run
tolerance	steps	Figure of merit	Figure of merit	Figure of merit	Figure of merit
0.1	5	0.0185	0.0231	0.0241	---
0.1	10	0.0185	0.0231	0.0241	0.0634
0.1	20	0.0185	0.0231	0.0241	0.0634
0.1	30	0.0185	0.0231	---	---
0.01	5	0.0184	0.0215	0.0222	0.0594
0.01	10	0.0183	0.0212	0.0218	0.0589
0.01	15	---	0.0213	0.0218	0.059
0.01	19	---	---	---	0.059
0.01	20	0.0182	0.0212	0.0218	0.0591
0.01	21	0.0186	---	---	0.059
0.01	22	0.0182	---	---	---
0.01	23	0.0186	---	---	---
0.01	24	0.0182	---	---	---
0.01	25	0.0186	---	0.0221	0.059
0.01	26	0.0182	---	---	---
0.01	27	0.0186	---	---	---
0.01	28	0.0182	---	---	---
0.01	29	0.0186	---	---	---
0.01	30	0.0182	0.0212	0.0221	0.0591
0.01	50	0.0182	0.0212	---	---
0.01	100	0.0182	---	0.0218	---
0.05	100	---	---	0.0224	---
0.005	100	---	---	0.0218	---
0.001	100	---	---	0.0218	---

10.4.2 Convergence of the Error Minimization Routine

In this section, two sample cases from the same successful estimation run are provided to illustrate the typical behavior of the algorithm. The first case, illustrated in Figure 10.11, demonstrates the most often encountered case in which the convergence tolerance is met in only one or two steps. Recall that the initial guess for the Gauss-Newton routine is the quaternion (0,0,0,1) for the first step, and the previous best estimate for all subsequent steps. Here it is clear that the error goes to some very small value after only two steps. Many times this convergence will occur in a single step. Figure 10.12 illustrates the corresponding quaternion elements at each time step for the same case. In fact, this is the case for a

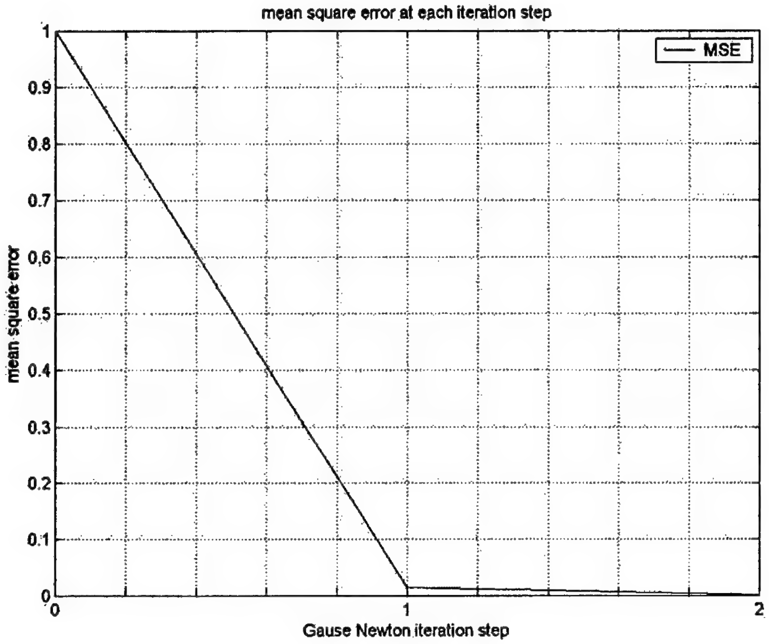


Figure 10.11: Single Time Step Error History for Successful Convergence

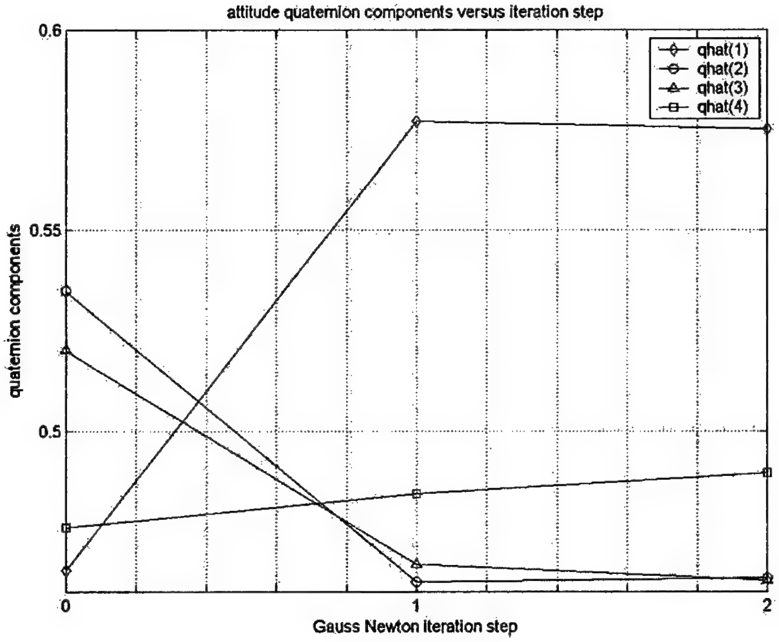


Figure 10.12: Single Time Step Quaternion Component History for Successful Convergence

rotation of 120 degrees about an axis of (1, 1, 1). In Section 10.1.4 it was demonstrated that this rotation corresponds to a rotation quaternion of (.5, .5, .5, .5). Here we see that, in this successful convergence case, the estimated quaternion is close to (.5, .5, .5, .5), but not exact. This is due to the error in the sensor measurements. While it at first may not appear very close, this combination of quaternion elements meets the mean square error convergence criteria, and is considered a "converged" case. The next two figures demonstrate the case where, as described above, the convergence criterion is not successfully met. These two figures are from the same run as the two previous figures, but earlier, before the rotation had approached 120 degrees. It is clear from these figures that increasing the number of iterations will not result in a significant reduction in error, since the mean square error has no slope and will not decrease further. This case demonstrates the utility of setting a relatively small maximum for the number of iterations allowed, thereby reducing run time of the routine. In fact, the majority of the error minimization occurs during the first few iterations, and even though the MSE convergence criterion is not met, a useful estimate of the attitude quaternion is produced and is available for input to the filter algorithm.

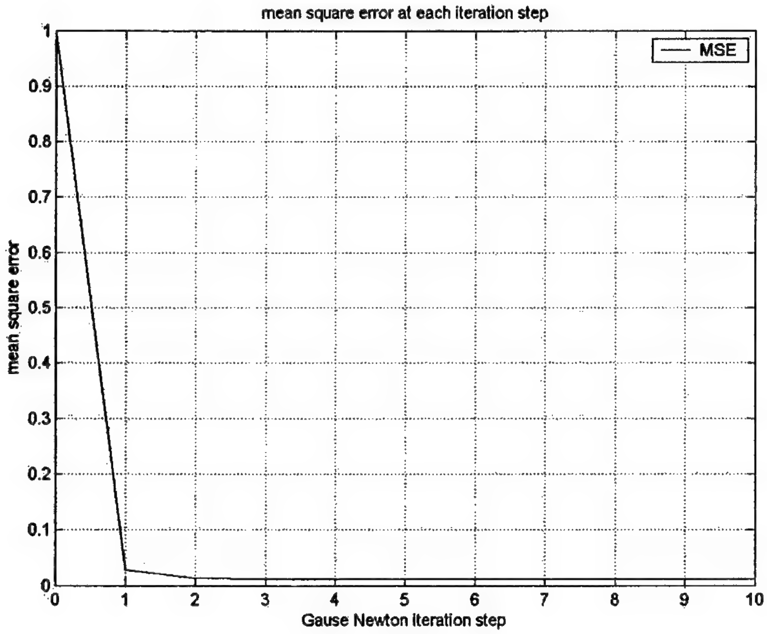


Figure 10.13: Single Time Step Error History for Unsuccessful Convergence

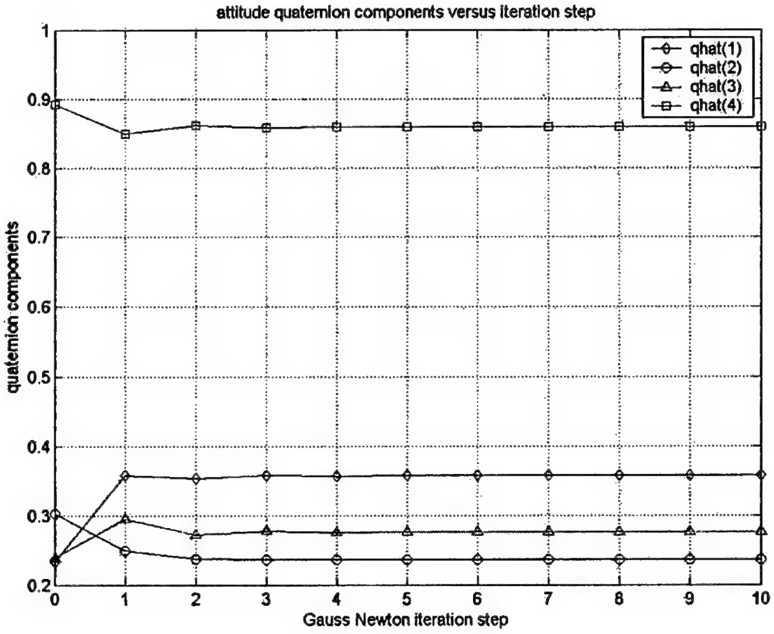


Figure 10.14: Single Time Step Quaternion Component History for Unsuccessful Convergence

10.5 Demonstrated Performance of the Extended Kalman Filter

Every filtering algorithm has a number of parameters that must be set to appropriate values for optimum operation of the filter. This process of determining the appropriate values and setting them in the algorithm is often referred to as "tuning" the filter. As described in Chapters 8 and 9, the EKF has several such values that must be determined. This section will discuss in more detail how these values are arrived at, and what values are used for the EKF designed in Chapter 9. Later in this section, the proper operation of the EKF will be demonstrated via examples illustrating performance both with and without the inclusion of sensors that measure rotational rates.

10.5.1 "Tuning" the Extended Kalman Filter

The extended Kalman filter designed in Chapter 9 has a number of parameters whose value must be determined for optimum operation of the filter in order to achieve the lowest possible error in the state estimate. As noted earlier, formulating the Measurement Noise Matrix, *R*, depends upon an accurate representation of $\bar{v}$, the vector containing the measurement standard deviations. The values for these were addressed in earlier sections, and these standard deviations are user-selectable in the software routine attitude_sim.m. In addition to these noise figures which may be ascertained through testing, specifications, etc., there are time constant parameters embedded within the process model illustrated in Figure 9.3., and there is process noise that goes into the calculation of the Process Noise Matrix, Q, as indicated in equations (9.36) and (9.38). As mentioned earlier, one of the drawbacks of using "optimal" filters is the somewhat complex design and tuning process. Several authors have made reference to the fact that "the approach is generally one of analysis rather than synthesis, and a number of steps are necessary before a satisfactory result emerges [71]," and that "Kalman filter design is both science and art, and different applications and operating environments call for grossly different Kalman filters [86]." The next few sections will attempt to shed some light on how this process plays out for this filter and application.

10.5.1.1 Choosing an Integration Scheme

Embedded within the EKF structure is a routine for the numerical integration of the nonlinear dynamic equations. Two choices must be made with respect to this integration. First, the quickest acceptable method is desired so as to minimize run time in an effort to yield an algorithm that can be implemented in real time. "Acceptable" is defined, here, as having sufficient accuracy such that errors due to numerical integration do not become significant. Second, the optimum step size must be determined, again balancing the tradeoff between algorithm speed and accuracy. As noted in [17], many numerical integration techniques exist for integrating differential equations. Through many examples, Zarchan demonstrates that a second-order Runge-Kutta technique works well, producing accurate answers without undue computational load. This method is used almost exclusively within [17] for the propagation of states due to its simplicity, ease of programming and because of the author's extensive successful experience with it. In at least one example, however, Zarchan points out that similar results are sometimes achieved using the simpler Euler integration method with the addition of process noise to account for the less accurate propagation [17]. In this work, extensive simulation with both a single step Euler integration and the second-order Runge-Kutta routines was conducted, and the more complex routine does not improve performance. In fact, due to the highly simplified model, the Euler method is the better performer, at all time steps. Therefore, in order to decrease the computation time due to numerical integration, the single step Euler method was implemented. The second question posed above is what step size to use. This requires some experimentation, because again there is a desire to use the largest "acceptable" step size in order to minimize run time. Zarchan describes one approach that is somewhat more structured than simply guessing a time step:

> The integration step size h must be small enough to yield answers of sufficient accuracy. A simple test, commonly practiced among engineers, is to find the appropriate integration

> step size by experiment. As a rule of thumb the initial step size is chosen to be several times smaller than the smallest time constant in the system under consideration. The step size is then halved to see if the answers change significantly. If the new answers are approximately the same, the larger integration step size is used to avoid excessive computer running time. If the answers change substantially, then the integration interval is again halved and the process is repeated [16].

Following a similar approach, the following sample results were generated experimentally for one of the EKF routines. The numbers in the body of the table are the average mean square error for the rotated reference vector and the overall figure of merit, as described earlier, for the run. The lower the figure of merit is, the better the estimation of attitude and rates.

Table 10.2: Sample Experimental Data for Determining Optimum EKF Integration Step Size

Integration Step Size	Rotated Vector Average MSE	Overall Figure of Merit
0.1 sec	.0110	.0143
0.01 sec	9.72e-4	.0011
0.001 sec	9.72e-4	.0011
0.0001 sec	9.72e-4	.0011

This is one example of the simulation runs accomplished to determine the best step size for the numerical integration of the nonlinear dynamic equations used to propagate the system states forward. These results are somewhat unexpected. Typically, a shorter time step will produce a better result when numerically integrating a function. What is demonstrated here, however, is that further decreasing the time step does not improve overall performance. This may be attributed to the very rudimentary model embedded within the filter from which the dynamic equations are generated. Its sole purpose is to propagate the states forward one time step, within the model, and the key criteria is that the time step is short relative to the time basis of the motion. As noted earlier, for the cases simulated here, the highest frequency motion is the 225 rev/min motion about the longitudinal axis of the rocket, and this corresponds to a time scale of approximately .013 sec. Based on the performance measures seen in the table, it is clear that decreasing the step size beyond 0.01 seconds does not improve performance. This assumes, of course, that the time interval at which measurements are available is not less than 0.01 seconds. In cases where the interval is smaller, a step size of shorter duration must be used for optimum results. For the cases simulated here, to minimize run time, 0.01 seconds is used in the Euler integration routine.

10.5.1.2 Setting Process Noise, Φ_S

Process noise in a Kalman type filter is a reflection of the uncertainty in the process model implemented by the filter. This can be very difficult to ascertain analytically, and even in cases where the model is known to be perfect, there can be reasons to artificially inflate the process noise. In cases where the system dynamics are known to be imperfectly modeled, it is often "advantageous to over-estimate the system noise so that Q_k is larger than strictly necessary [34]." As noted previously, the process noise is used to initialize the Process Noise Matrix and appears in the computation of the Discrete Process Noise Matrix at each iteration of the filter. Adding fictitious process noise to the actual estimate of the uncertainty in the model is a commonly accepted engineering solution for many of the problems encountered when implementing EKFs. One such example is when divergence is encountered in the error in the estimate. One cause of such divergence is inaccurately propagating the states forward in time. The obvious solution is to choose a more accurate numerical integration technique, or a smaller time step. Either approach results in greater computational load. Another solution, often employed, is the addition of process noise to the filter [17], [75]. In many cases, this will prevent the divergence while allowing the use of a less computationally intense integration method and step size. The price to be paid, however, is typically a noisier estimate of the state than would be achieved with a more accurate propagation [17]. Put another way, "increasing the process noise effectively increases the bandwidth of the filter, which improves its tracking capabilities at the expense of more noise transmission [17]." From this perspective, increasing the process noise, even artificially, increases the bandwidth of the filter and makes the filter less sluggish, while a lower process noise might cause the filter to lag the actual signal [17]. In this way, an EKF is like any other filter, higher bandwidth often means quicker response, but also allows in more noise. As a practical matter, setting the process noise too low "eventually causes the filter to stop paying attention to new measurements (i.e., filter bandwidth is reduced or the filter gain is too small for incorporating more measurements). In other words, a zero process noise extended Kalman filter eventually goes to sleep [17]!"

For this application, the process model is known to be a simplified version of the real processes producing rotational motion of the rocket. This simplification is intentional both because the true model is very complex, and so that one model is easily transportable between various vehicles. As a result, the process noise is known to be non-zero. As is usually the case, for the reasons noted above, the analytical determination of how much process noise to add is very difficult. As an alternative, the optimum process noise for the algorithms developed here is determined empirically. Admittedly, this is a luxury of working with simulated data, where the "truth" is known and where the same data may be processed repeatedly to determine the value of process noise that produces the best performance. Each algorithm allows the user to enter the amount of process noise to be used for that filter run, therefore this can be easily varied to evaluate the results corresponding to different amounts of process noise while holding all else equal.

This procedure yields the results in the table below for a baseline example consisting of a profile with rotational rates that asymptotically approach final values of 0.5 rev/min about the body x-axis, 0.5 rev/min about the body y-axis, and 225 rev/min about the body z-axis. In each case the profile achieves its final value at 20 seconds into the run with a total simulation time of 40 sec. This simulation used the baseline sensor standard deviations described in Chapter 6 of 1.333 degrees in each axis for the Sun sensor, 3.333 degrees in each axis for the magnetometer, and 0.333 rev/min for the rate sensors. Following this procedure, the following data was generated for the EKF using measured rates.

Table 10.3: Sample Experimental Data for Determining Optimum Process Noise for EKF

Φ_S	$\tau_{\omega_1} = \tau_{\omega_2}$	τ_{ω_3}	EKF Average MSE of Rotated Reference Vector	EKF Figure of Merit
36	36	1e7	.0010	.0012
38	36	1e7	.0010	.0011
⋮	⋮	⋮	⋮	⋮
112	36	1e7	9.72e-4	.0011
⋮	⋮	⋮	⋮	⋮
185	36	1e7	9.68e-4	.0011
190	36	1e7	9.68e-4	.0012

This is a small subset of the data generated for one run using one algorithm, the EKF with rate sensor data. Values outside this range were tested, all with higher figures of merit. This is meant to show a "close-up" of the range of values producing the optimum results. An example of a complete set is found in the appendices. Here it is evident that the "tuning" process is not exact. In fact, process noise values of 38 – 185 yield almost identical results. While this makes the EKF "robust" with respect to different values of entered process noise, it also makes it difficult to pin down an optimum number. For a broad range of values that yield similar performance, such as those found here, a median value is typically selected as the optimum. For this particular case, the value selected is 112. By comparison, the range of values found when no rate measurements are used is 3.42 – 3.64, and an "optimized" value of 3.53 is used.

10.5.1.3 Setting Motion Time Constants, τ_r

As with the process noise discussed in the previous section, the motion time constants from the process model may be entered by the user when running the software that implements the various algorithms developed here. This allows the time constants τ_{ω_1}, τ_{ω_2}, and τ_{ω_3} to be varied one at a time while holding all other parameters equal to evaluate their impact on the performance of the algorithm, and to determine optimum values for a particular set of data. In the case where actual data is being post-processed, these constants may be determined using an automated routine, such as that employed in [11], where the process model is simulated in SIMULINK, a MATLAB utility, and the time constants and noise variances are adjusted until the best match with the real data is achieved. A similar approach is used here for the simulated data, although a manual iteration of runs was accomplished. In this case, the optimum process noise is determined using the approach described in the last section. Then, using this optimum process noise, the time constants are varied until the set producing the overall minimum figure of merit is discovered. Since the x-axis and y-axis rate profiles in the baseline trajectory are identical, the time constants for these two are set equal. An example of this process, for the baseline trajectory described in the last section, but now with no rate measurements used, is illustrated by the sample data in the following table.

Table 10.4: Sample Experimental Data for Determining Optimum EKF Time Constants

Φ_S	$\tau_{\omega_1} = \tau_{\omega_2}$	τ_{ω_3}	EKF Average MSE of Rotated Reference Vector	EKF Figure of Merit
3.53	23	1e7	.0010	.0689
3.53	24	1e7	.0010	.0688
⋮	⋮	⋮	⋮	⋮
3.53	29	1e7	.0010	.0688
⋮	⋮	⋮	⋮	⋮
3.53	33	1e7	.0010	.0688
3.53	34	1e7	.0010	.0689

Again, this is a small subset intended to show a close-up of the range of interest, and a complete example of the data generated is found in the appendices. Note that the optimized Φ_s in this case is different than for the example in Section 10.5.1.2, because this is for a case where differenced rates are used and the earlier example came from a case where rate measurements are used. From this excerpt, the optimized value for $\tau_{\omega_1} = \tau_{\omega_2}$ is 29, again choosing a median value from a range yielding similar performance as determined by overall figure of merit.

For all the runs accomplished in this work, the time constants, and the process noise, are optimized empirically to provide the best possible figure of merit to allow for the side-by-side comparison of relative performance between algorithms. While these tuning parameters are undoubtedly functions of the dynamics of the motion, they are also clearly functions of sensor measurement quality and the effect of the suboptimal dynamic model embedded within the filter. This is made evident by the fact that the optimum time constants, and process noise, are different for each of the four algorithms tested, even when processing identical data. No analytical relationship was developed for determining these tuning parameters. In every case they were optimized through the iterative process described in this section and, for the process noise, in the previous section. Clearly, such an analytical expression would be a very useful tool from an implementation standpoint, and warrants further investigation.

10.5.2 Sample Performance When no Rate Measurements are Available

As described in Chapter 9, the EKF has been implemented to work with and without actual rate sensor measurements. This section describes the performance of the filter when these rate measurements are not available and this filter is implemented in norates_ekf.m found in Appendix A. Of interest from an implementation standpoint, this method requires the differencing of subsequent quaternions to generate a quaternion rate. This in turn is used to calculate corresponding body rates as developed in equations (9.2) and (9.3). Following this approach obviously incurs a one time step delay before estimates may be generated. For small time steps, however, this should not be a major impact although it must be evaluated for any projected application. If such a delay is acceptable, as it certainly would be in a post-processed application, then a minimum sensor suite composed only of two vector sensors may be employed, without the need for rate sensors. The following table illustrates the type of performance available from this implementation. This data was generated for the duration of simulation shown using sensor measurement standard deviations of 1.333 degrees for each axis of the Sun sensor and 3.333 degrees for the magnetometer measurements. The rate profiles asymptotically approach the final values of 0.5 rev/min, 0.5 rev/min, and 225 rev/min about the body x, y, and z axes respectively. The final values are achieved at 20 seconds. The integration step size is the standard 0.01 seconds. Where, again, Φ_S is the process noise, $\tau_{\omega_1} = \tau_{\omega_2}$ is the time constant associated with ω_1 and ω_2 and τ_{ω_3} is the time constant associated with ω_3. The MSE and the figure of merit are as defined earlier. Table 10.6 displays a number of "angle"

Table 10.5: Sample EKF Performance When no Rate Measurements are Available

Duration	Φ_S	$\tau_{\omega_1} = \tau_{\omega_2}$	τ_{ω_3}	EKF Average MSE of Rotated Reference Vector	EKF Figure of Merit
40	3.53	29	1e7	.0010	.0688
40	3.57	28	1e7	9.71e-4	.0679
60	3.10	42	1e7	9.94e-4	.0571
60	3.09	42	1e7	9.78e-4	.0587

Table 10.6: EKF Angle Measure Performance When no Rate Measurements are Available

$\text{Mean}_{\text{Angle1}}$	σ_{Angle1}	$\text{Max}_{\text{Angle1}}$	$\text{Mean}_{\text{Angle2}}$	σ_{Angle2}	$\text{Max}_{\text{Angle2}}$	$\text{Mean}_{\text{Angle3}}$	σ_{Angle3}	$\text{Max}_{\text{Angle3}}$
N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
2.75	1.68	12.69	2.76	1.73	12.30	3.23	1.92	13.61
N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
2.80	1.68	11.44	2.78	1.69	13.21	3.27	1.89	12.21

performance measures that correspond to these numerical measures of performance. While the figure of merit defined earlier is the adopted standard of performance for relating one method to another for a given set of data, these angle measures help give an intuitive feel for how well the filter is performing. Angle1 refers to the angle between the body x axis rotated using the "true" quaternion and the body x-axis rotated using the filter estimated quaternion, or the "error" angle for the first axis. Angle2 and Angle3 are similarly calculated for the body y and z axes respectively. σ indicates the standard deviation of the error angle. The rows in this table correspond to the rows in the previous table, coming from the same simulated data. Any table entries of "N/A" indicate that those runs were accomplished during the initial "tuning" process, using the figure of merit, and the option to calculate angles was not selected.

10.5.3 Sample Performance When Rate Measurements are Available

The EKF has also been implemented to accept actual noisy rate measurements from rate sensors. This implementation is accomplished in rates_ekf.m found in Appendix A. This section describes the performance of the filter when these rate measurements are available. As noted in [13], and demonstrated here, the use of rate sensor data eliminates much of the "lag" experienced when estimating the rotational rates found in the filter that does not incorporate rate measurements. Use of these sensors also eliminates the one time step delay before estimates become available. These two improvements come at the expense of greater cost and complexity to include rate sensors to measure rates about each of the body axes. The following table illustrates the type of performance available from this implementation. This data was generated for the duration of simulation shown using sensor measurement standard deviations of 1.333 degrees for each axis of the Sun sensor and 3.333 degrees for the magnetometer measurements. The rate profiles asymptotically approach the final values of 0.5 rev/min, 0.5 rev/min, and 225 rev/min about the body x, y, and z axes respectively. The final values are achieved at 20 seconds. The integration step size is the standard 0.01 seconds.

Table 10.7: Sample EKF Performance When Rate Measurements are Available

Duration of Simulation (seconds)	Φ_S	$\tau_{\omega_1} = \tau_{\omega_2}$	τ_{ω_3}	EKF Average MSE of Rotated Reference Vector	EKF Figure of Merit
40 (set #1)	112	36	1e7	9.72e-4	.0011
40 (set #2)	122	36	1e7	9.49e-4	.0011
60 (set #3)	175	36	1e7	9.77e-4	.0011
60 (set #4)	175	36	1e7	9.62e-4	.0011

10.6 Demonstrated Performance of the Unscented Kalman Filter

As is the case for the extended Kalman filter, the unscented Kalman filter has a number of parameters that must be set to appropriate values for optimum operation of the filter. This section will discuss how these values are arrived at, and what values are used for the UKF that was designed in Chapter 9. Later in this section, the proper operation of the UKF will be demonstrated via examples illustrating performance both with and without the inclusion of sensors that measure rotational rates.

Table 10.8: EKF Angle Measure Performance When Rate Measurements are Available

$\text{Mean}_{\text{Angle1}}$	σ_{Angle1}	$\text{Max}_{\text{Angle1}}$	$\text{Mean}_{\text{Angle2}}$	σ_{Angle2}	$\text{Max}_{\text{Angle2}}$	$\text{Mean}_{\text{Angle3}}$	σ_{Angle3}	$\text{Max}_{\text{Angle3}}$
N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
2.72	1.65	12.61	2.73	1.71	12.26	3.19	1.91	13.59
N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
2.77	1.67	11.47	2.74	1.68	13.12	3.24	1.88	12.23

10.6.1 "Tuning" the Unscented Kalman Filter

The unscented Kalman filter incorporates all the same parameters as the EKF, but also has several additional values that must be determined and set in order to achieve the lowest possible error in the state estimate. As for the EKF, formulating the Measurement Noise Matrix, R, in the UKF depends upon an accurate representation of $\bar{v}$, the vector containing the measurement standard deviations. These values were addressed in earlier sections, and these standard deviations are user-selectable in the software routine attitude_sim.m. In addition to these noise figures which may be ascertained through testing, specifications, etc., there are time constant parameters corresponding to each of the rotational rates embedded within the process model illustrated in Figure 9.3., and there is process noise that goes into the calculation of the Process Noise Matrix, Q, as indicated in equations (9.51) and (9.52). As for the EKF, the process for optimizing these values for the UKF is not an easy one. The next few sections will detail their determination.

10.6.1.1 Choosing an Integration Scheme

As was the case with the EKF, the UKF employs numerical integration to propagate states forward in time. In this case, however, the entire augmented set of "sigma" points described in Section 9.8.2.9 must be propagated through the nonlinear system dynamic equations. Since there are $2(2L)+1$ points in the augmented set, or a total of 29 in this case, choosing an efficient numerical technique is very important to minimize run time. As was done with the EKF, both a 2nd order Runge-Kutta and a one-step Euler method were evaluated. The Euler method is the better performer in addition to requiring fewer computational steps. Because the same motion is simulated for the UKF and an identical suboptimal dynamic model is used to generate the state equations, the discussion in Section 10.5.1.1 applies here as well. Extensive simulation with different time steps was accomplished and an optimum time step of .01 sec was determined, for the cases simulated here. The following table displays one such set of results for the Euler method. From the tabulated data, it is clear that the .01 sec step size achieves the desired result while minimizing the number of calculations.

Table 10.9: Sample Experimental Data for Determining Optimum UKF Integration Step Size

Integration Step Size	Rotated Vector Average MSE	Overall Figure of Merit
0.1 sec	.0831	.0831
0.01 sec	4.48e-4	6.37e-4
0.001 sec	4.48e-4	6.37e-4
0.0001 sec	4.48e-4	6.37e-4

10.6.1.2 Setting Process Noise, Φ_S

Setting process noise for the UKF is subject to the same guidelines discussed in Section 10.5 for the EKF. Here again, additional process noise is often used to cover for uncertainties in the system model, noise parameters, etc. An analytical determination is intractable in this case, so as was done for the EKF, the optimum noise is determined through an iterative, empirical process. The process noise is varied in small increments and the overall performance of the filter is evaluated at each value. The minimum "figure of merit" corresponds to the optimum value for the process noise. The following is an excerpt of one such application of the process for the baseline trajectory.

Table 10.10: Sample Experimental Data for Determining Optimum Process Noise for UKF

Φ_S	$\tau_{\omega_1} = \tau_{\omega_2}$	τ_{ω_3}	UKF Average MSE of Rotated Reference Vector	UKF Figure of Merit
1.09	.07	1e7	.0010	.0450
1.08	.07	1e7	.0010	.0449
1.07	.07	1e7	.0010	.0449
1.06	.07	1e7	.0010	.0449
1.05	.07	1e7	.0010	.0449
1.04	.07	1e7	.0010	.0450

For this particular example case, it is evident that values of process noise between 1.04 and 1.09 yield the best overall performance. A broader range of values was tested, and this table shows a "close-up" of the range of interest. Given these results, a value of 1.07 is used as the optimized process noise. This is one example of determining the process noise. Making the choice of this parameter more difficult, in a general sense, is the fact that the "optimum" value varies with changes in sensor accuracy, body rate profile, and is affected by whether the UKF uses measured rates or rates from differencing. This parameter must be "tuned" for the best estimate of the application motion characteristics. As noted in section 10.5, no analytical expression was determined and this is an area that warrants further research.

10.6.1.3 Setting the Time Constants of Motion, τ_r

A method similar to that for determining the process noise is employed for the time constants embedded in the process model. Again, in the case where actual rotational data is available, the variances and time constants may be determined through post-processing using an automated, iterative procedure as that used by Marins [11] and described in Section 10.5. Here, a manual iterative process is used where one time constant is varied while all other parameters are held constant. Once a minimum figure of merit is achieved, the optimized time constant is determined. Since the ω_1 and ω_2 profiles are identical, the time constants associated with these are set equal to each other. This is a luxury of using simulated data, where the same data may be processed again and again while varying one parameter at a time. A similar approach could be used with real data if the flight was post-processed, but for a real-time approach a best estimate would have to be used or other measures taken to predict the optimum time constants associated with the motion. An excerpt of the data generated for the baseline trajectory is shown in Table 10.11. For this particular example, choosing a median value from the range producing similar results gives an optimized time constant for ω_1 and ω_2 of 0.07. A similar process would then be completed for determining τ_{ω_3}.

10.6.1.4 Choosing the UT Parameters

Thus far, tuning the UKF has been identical to tuning the EKF, although different optimum parameter values are found for each when processing the same data. In addition to the process noise, and the three time constants, there are several other parameters that differ from those found in the EKF. These were identified in Chapter 9 in equations (9.56) and (9.61) for the original set of sigma points, and in (9.63) and (9.64) for the augmented set. The first of these, α, is a scaling parameter that determines the spread of the sigma points about the mean value for the state, and is usually set to a small positive value (e.g., $10^{-4} \leq \alpha \leq 1$)

Table 10.11: Sample Experimental Data for Determining Optimum UKF Time Constants

Φ_S	$\tau_{\omega_1} = \tau_{\omega_2}$	τ_{ω_3}	UKF Average MSE of Rotated Reference Vector	UKF Figure of Merit
1.07	.04	1e7	.0010	.0451
1.07	.05	1e7	.0010	.0449
⋮	⋮	⋮	⋮	⋮
1.07	.07	1e7	.0010	.0449
⋮	⋮	⋮	⋮	⋮
1.07	.10	1e7	.0010	.0449
1.07	.11	1e7	.0010	.0450

[68]. Not much guidance exists on the selection of a value for this parameter, but as pointed out by [68], the value of α is not critical in state estimation applications of the UKF, which is how it is employed here. Therefore, α is set to a value of .01. The insensitivity to changes in α was confirmed through simulation and the following table illustrates an example of the results. The β parameter corresponds to the type of distribution expected of the data. Given no information to the contrary, this distribution is typically

Table 10.12: UKF Performance Sensitivity to Changes in α and β Parameters

α	β	Φ_S	$\tau_{\omega_1} = \tau_{\omega_2}$	τ_{ω_3}	UKF Average MSE of Rotated Reference Vector	UKF Figure of Merit
1e-4	2	.023	.20	1e7	4.48e-4	6.37e-4
.05	2	.023	.20	1e7	4.48e-4	6.37e-4
.1	2	.023	.20	1e7	4.48e-4	6.37e-4
.5	2	.023	.20	1e7	4.48e-4	6.37e-4
1	2	.023	.20	1e7	4.48e-4	6.37e-4
10	2	.023	.20	1e7	4.48e-4	6.37e-4
.01	1	.023	.20	1e7	4.48e-4	6.37e-4
.01	5	.023	.20	1e7	4.48e-4	6.37e-4

assumed to be Gaussian, and a value of $\beta=2$ is optimum for a Gaussian distribution [68]. The EKF has no such parameter, and if the sensor or process noise is indeed non-Gaussian, there may a more suitable value of β allowing the UKF to achieve better results for the non-Gaussian case. Little research has been done in this area, and it warrants further investigation. As Holt points out, "In the practical case there is usually no *a priori* reason for an assumption of a non-Gaussian distribution [34]." That assumption is made for this work, and $\beta=2$ is used in all cases. As evidenced by the data in the table above, however, in this application the filter is insensitive to changes in this parameter, at least over the range tested. A third parameter that is needed for implementation of the UKF is κ, as found in equation (9.56). This is a secondary scaling parameter that is typically set to $3-L$ where L is the dimension of the state vector [68]. In this case $L = 7$, and the parameter κ is set to -4 for the original sigma set.

If an augmented set of sigma points is used, as it is in this design, the parameters α', β', and κ' must also be determined. These correspond to the parameters α, β, and κ described above for the unaugmented set, and the same guidelines apply for setting their values. For this effort they are set to, $\alpha'=.01$, $\beta'=2$, and $\kappa'=-11$.

10.6.2 Sample Performance When no Rate Measurements are Available

As with the EKF described in Section 10.5, the UKF has been implemented both to accept actual noisy rate measurements from rate sensors and to operate on "measurements" derived from differencing quaternion rates as described in equations (9.1) and (9.2). This section discusses performance when no direct measurements of rate are available. As with the EKF without rates, the differencing causes a one time step delay in the processing. As was mentioned earlier, for most applications, and assuming a small time step, this should not be an issue. The software to implement this filter is norates_ukf.m and is found in Appendix A. The data in the following table was generated for the duration of simulation shown using sensor measurement standard deviations of 1.333 degrees for each axis of the Sun sensor and 3.333 degrees for the magnetometer measurements. The rate profiles asymptotically approach the final values of 0.5 rev/min, 0.5 rev/min, and 225 rev/min about the body x, y, and z axes respectively. The final values are achieved at 20 seconds. The integration step size is the standard 0.01 seconds. These results are from processing the identical data sets as processed for the EKF results in Section 10.5, with the rows from all tables corresponding to the same simulation data. This was done to facilitate straight-across comparison of performance between filter algorithms. In each case, the filter parameters have been tuned to achieve the best possible figure of merit for the particular data set, and the table values represent the optimum performance for the filter, as implemented here, on that set of simulated data. Here, again, Φ_S is the process noise, $\tau_{\omega_1} = \tau_{\omega_2}$ is the time constant associated with ω_1 and ω_2 and τ_{ω_3} is the time constant associated with ω_3. The MSE and the figure of merit are as defined earlier.

Table 10.13: Sample UKF Performance When no Rate Measurements are Available

Duration of Simulation (seconds)	Φ_S	$\tau_{\omega_1} = \tau_{\omega_2}$	τ_{ω_3}	UKF Average MSE of Rotated Reference Vector	UKF Figure of Merit
40 (set #1)	1.07	.07	1e7	.0010	.0449
40 (set #2)	1.05	.09	1e7	9.73e-4	.0440
60 (set #3)	1.01	.07	1e7	9.96e-4	.0373
60 (set #4)	0.99	.07	1e7	9.80e-4	.0376

The following table displays a number of "angle" performance measures that correspond to these numerical measures of performance. While the figure of merit defined earlier is the standard of performance for relating one method to another for a given set of data, these angle measures help give an intuitive feel for how well the filter is performing. Angle1 refers to the angle between the body x axis rotated using the "true" quaternion and the body x-axis rotated using the filter estimated quaternion, or the "error" angle for the first axis. Angle2 and Angle3 are similarly calculated for the body y and z axes respectively. σ indicates the standard deviation of the error angle. The rows in this table correspond to the rows in the previous table, coming from the same simulated data.

Table 10.14: UKF Angle Measure Performance When no Rate Measurements are Available

$\text{Mean}_{\text{Angle1}}$	σ_{Angle1}	$\text{Max}_{\text{Angle1}}$	$\text{Mean}_{\text{Angle2}}$	σ_{Angle2}	$\text{Max}_{\text{Angle2}}$	$\text{Mean}_{\text{Angle3}}$	σ_{Angle3}	$\text{Max}_{\text{Angle3}}$
N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
2.76	1.68	12.69	2.76	1.73	12.30	3.23	1.92	13.60
N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
2.80	1.68	11.44	2.78	1.70	13.21	3.28	1.89	12.20

10.6.3 Sample Performance When Rate Measurements are Available

Finally, the UKF has been implemented to accept actual noisy rate measurements from rate sensors in rates_ukf.m found in Appendix A. Using rate sensors eliminates the lag seen when estimating rotational rates from differencing and results in better estimation of both the attitude quaternion and the rotational rates. An example of this lag is seen in the next two figures. These figures are from a simulation of the baseline case, and the lag in the estimate for the "no rates" filter is clearly seen in the first figure, and not present in the "with rates" filter results. The price to be paid is increased cost and complexity in order to incorporate the rate sensors for each of the body axes into the system. The data in the following table was generated for the duration of simulation shown using sensor measurement standard deviations of 1.333 degrees for each axis of the Sun sensor and 3.333 degrees for the magnetometer measurements. The rate profiles asymptotically approach the final values of 0.5 rev/min, 0.5 rev/min, and 225 rev/min about the body x, y, and z axes respectively. The final values are achieved at 20 seconds. The integration step size is the standard 0.01 seconds. These results are from processing the identical data sets as processed for the EKF results in Section 10.5 and for the UKF without rate measurements in the last section. The rows from

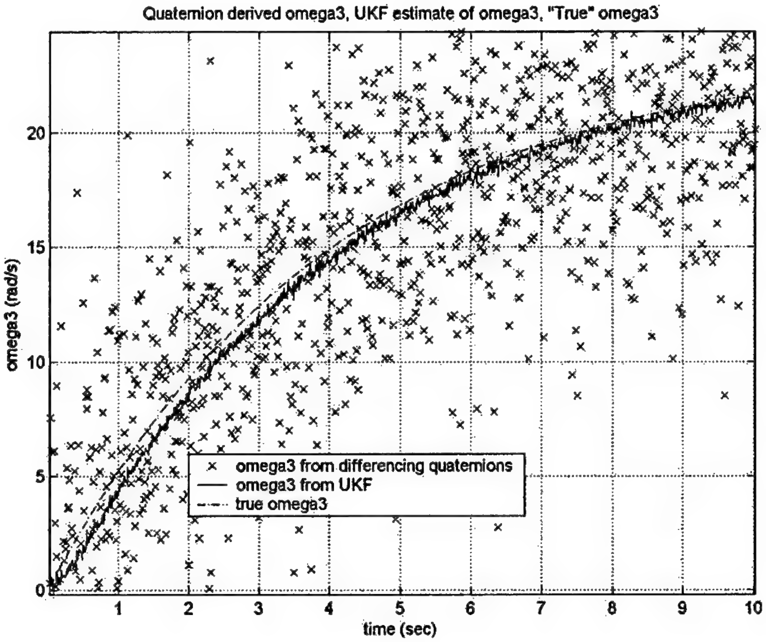


Figure 10.15: Lag in Filter Rate Estimate When Rate Measurements not Used

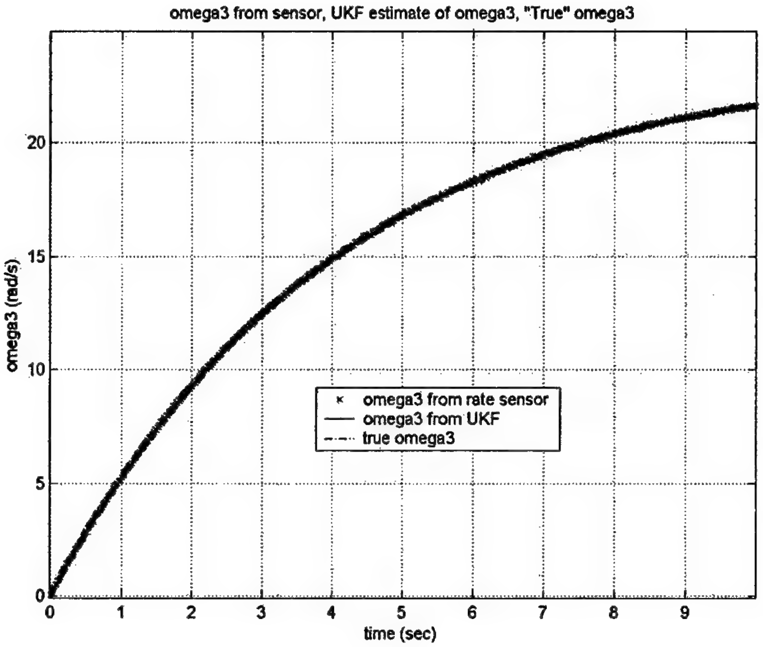


Figure 10.16: No Lag in Filter Rate Estimate When Rate Measurements are Used

all tables correspond to the same simulation data. Again, straight-across comparison of performance between filter algorithms is possible. In every case, the filter parameters have been tuned to achieve the best possible figure of merit for the particular data set, and the table values represent the optimum

Table 10.15: Sample UKF Performance When Rate Measurements are Available

Duration of Simulation (seconds)	Φ_S	$\tau_{\omega_1} = \tau_{\omega_2}$	τ_{ω_3}	UKF Average MSE of Rotated Reference Vector	UKF Figure of Merit
40 (set #1)	.023	.20	1e7	4.48e-4	6.37e-4
40 (set #2)	.023	.36	1e7	4.41e-4	6.34e-4
40 (set #5)	.023	.32	1e11	4.43e-4	6.30e-4
60 (set #3)	.020	.50	1e7	3.78e-4	5.94e-4
60 (set #4)	.022	.47	1e7	4.23e-4	5.85e-4

performance for the filter, as implemented in this work, on that set of simulated data. Here, again, Φ_S is the process noise, $\tau_{\omega_1} = \tau_{\omega_2}$ is the time constant associated with ω_1 and ω_2 and τ_{ω_3} is the time constant associated with ω_3. The MSE and the figure of merit are as defined earlier. The Table 10.16 displays a number of "angle" performance measures that correspond to these numerical measures of performance. While the figure of merit defined earlier is the adopted standard of performance for relating one method to another for a given set of data, these angle measures help give an intuitive feel for how well the filter is performing. Angle1 refers to the angle between the body x axis rotated using the "true" quaternion and the body x-axis rotated using the filter estimated quaternion, or the "error" angle for the first axis. Angle2 and Angle3 are similarly calculated for the body y and z axes respectively. σ indicates the standard deviation of the error angle. The rows in this table correspond to the rows in the previous table, coming from the same simulated data.

Table 10.16: UKF Angle Measure Performance When Rate Measurements are Available

$\text{Mean}_{\text{Angle1}}$	σ_{Angle1}	$\text{Max}_{\text{Angle1}}$	$\text{Mean}_{\text{Angle2}}$	σ_{Angle2}	$\text{Max}_{\text{Angle2}}$	$\text{Mean}_{\text{Angle3}}$	σ_{Angle3}	$\text{Max}_{\text{Angle3}}$
N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
1.84	1	6.55	1.86	1.05	7.21	2.01	1.10	7.12
1.82	1.01	7.35	1.85	1.04	7.93	2.06	1.12	7.70
N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
1.80	0.98	6.52	1.82	1.00	6.95	1.97	1.06	7.17

10.7 Summary of Filter Relative Performance

A small subset of the overall simulation data used to evaluate the various algorithms has been presented in this chapter. The objective was to determine which of the methods, those based on a more traditional extended Kalman filter approach or those based on the relatively new unscented Kalman filter, provides the greatest leverage with respect to the low-cost sounding rocket attitude problem. Some of the more important constraints associated with this problem are that it is nonlinear, that low-cost approaches often mean less sensors and that those sensors are of lower accuracy, and somewhat limited availability of computing resources. The performance data from the last several sections is compiled in the following table for comparison purposes. Added to what was presented before, is the performance achieved using the quaternions derived from the Gauss-Newton error minimization, without filtering. The measure GN-MSE is generated using this quaternion to rotate the reference vectors and then making a comparison with the reference vectors rotated using the "true" quaternion. These and the rotational rate "measurements," either measured or from differencing, are used to calculate the GN-FOM, or "Gauss-Newton Figure of Merit" in the same manner as the EKF and UKF figures of merit are calculated. With this new information, a true side-by-side comparison of the performance of each filter type, and the unfiltered solution, is made possible. As before, each row contains results from processing the identical data set by each method.

Table 10.17: Summary of Algorithm Relative Performance

	GN-MSE	EKF-MSE (no rates)	UKF-MSE (no rates)	EKF-MSE (with rates)	UKF-MSE (with rates)	GN-FOM (no rates)	EKF-FOM (no rates)	UKF-FOM (no rates)	GN-FOM (with rates)	EKF-FOM (with rates)	UKF-FOM (with rates)
Data Set #1	.0010	.0010	.0010	9.72e-4	4.48e-4	37.31	.0688	.0449	.0016	.0011	6.37e-4
Data Set #2	9.74e-4	9.71e-4	9.73e-4	9.49e-4	4.41e-4	35.56	.0679	.0440	.0016	.0011	6.34e-4
Data Set #3	9.97e-4	9.94e-4	9.96e-4	9.77e-4	3.78e-4	37.06	.0571	.0373	.0016	.0011	5.94e-4
Data Set #4	9.80e-4	9.78e-4	9.80e-4	9.62e-4	4.23e-4	35.86	.0587	.0376	.0016	.0011	5.85e-4

Evaluating the figures of merit for the various algorithms reveals that the UKF method using rate sensor data is significantly better than all others. In fact, the results from data set #4, one of the 60 second simulations, indicate that the figure of merit is improved by nearly 47 percent over the nearest competitor, the EKF using rate sensors! Clearly the performance of the methods that do not incorporate rate sensors is lower, as would be expected. It should be pointed out, however, that these algorithms are indeed successful at estimating all the states, albeit at a lower accuracy than the "with rates" algorithms. As is clearly shown in the table above, both the EKF and UKF based filters show huge improvement over simply using the Gauss-Newton error minimization directly and then differencing quaternions to estimate rates. The exceptionally large figures of merit for the Gauss-Newton case reflect the relatively poor overall performance achieved when no processing of the rate measurements has been done. Examination of the GN-MSE column, which does not assess estimation of the rates, indicates that the attitude determination performance is very good having only been processed by the Gauss-Newton routine. The question for a system designer then becomes one of trade-offs, balancing the increased performance of including rates against the simplicity and lower accuracy of a "no-rates" approach. Referring to set #4 again, the best "no rates" algorithm is the UKF again, displaying an approximately 36 percent improvement in figure of merit over the EKF method without rates.

Examination of the UKF-MSE (no rates) column at first appears to indicate an aberration in that the UKF-based algorithm does more poorly than the EKF-based algorithm with respect to the mean square error criteria. This is an artifact of tuning to the lowest figure of merit, and not to the lowest MSE. In this way the filter is optimized for estimating both the rotational rates and the attitude quaternion. In effect, this is a best compromise. In every case tested, a set of parameters were found that yielded a lower MSE for the UKF than that achievable by the EKF. This approach to tuning results in a higher overall figure of merit, but yields a slightly more accurate attitude solution. The choice of tuning approach, for best attitude or for best figure of merit, must be made by the user, taking into account the particular problem parameters (i.e., importance of attitude relative to rates, etc.).

While the figures of merit in the table above clearly indicate relative performance between filters, they do not provide an easily visualized, or intuitive, measure of performance. The next table summarizes the "error angles" associated with the solution from each algorithm. These angles are calculated between the body axes when rotated using the true quaternion and the body axes when rotated using the filter-produced quaternion. While they do not give an indication of the rate estimation performance, they do help visualize the relative performance of the different filters from a "pointing accuracy" perspective. Angle 1 corresponds to the first body axis, and Angle 2 and Angle 3 correspond to the second and third axes respectively Here again, the same relative performance is seen. The UKF filter using rates achieves much better results than does any other. From the tabulated data, it is clear that this filter produces lower mean errors and errors with a smaller standard deviation. Again, the nearest competitor is the EKF filter that uses rates, and the UKF filter enjoys a significant performance advantage. Evaluating the spin axis pointing error, angle 3, the UKF performance is approaching an improvement of 37 percent for data set #2 and is approximately 39 percent better for data set #4.

Another significant observation from these results is that the "no rates" methods do not yield much improvement in terms of pointing accuracy. This is likely due to the very simple embedded dynamic model and the fact that the rates derived from differencing are still very noisy. It seems clear that for real improvement in attitude estimation, rates must be included and, as demonstrated above, the UKF-based algorithm clearly outperforms the EKF.

Table 10.18: Summary of Algorithm Relative Error Angle Performance

Data Set #2 (40 sec)	***Mean Angle 1***	σ_{Angle1}	***Max Angle 1***	***Mean Angle 2***	σ_{Angle2}	***Max Angle 2***	***Mean Angle 3***	σ_{Angle3}	***Max Angle 3***
Gauss-Newton	2.76	1.68	12.70	2.76	1.73	12.31	3.23	1.92	13.61
EKF (no rates)	2.75	1.68	12.69	2.76	1.73	12.30	3.23	1.92	13.61
UKF (no rates)	2.76	1.68	12.69	2.76	1.73	12.30	3.23	1.92	13.60
EKF (with rates)	2.72	1.65	12.61	2.73	1.71	12.26	3.19	1.91	13.59
UKF (with rates)	1.84	1.00	6.55	1.86	1.05	7.21	2.01	1.10	7.12
Data Set #4 (60 sec)	***Mean Angle 1***	σ_{Angle1}	***Max Angle 1***	***Mean Angle 2***	σ_{Angle2}	***Max Angle 2***	***Mean Angle 3***	σ_{Angle3}	***Max Angle 3***
Gauss-Newton	2.80	1.68	11.44	2.78	1.70	13.22	3.28	1.89	12.21
EKF (no rates)	2.80	1.68	11.44	2.78	1.69	13.21	3.27	1.89	12.21
UKF (no rates)	2.80	1.68	11.44	2.78	1.70	13.21	3.28	1.89	12.20
EKF (with rates)	2.77	1.67	11.47	2.74	1.68	13.12	3.24	1.88	12.23
UKF (with rates)	1.80	0.98	6.52	1.82	1.00	6.95	1.97	1.06	7.17

Given the substantial performance advantages associated with the UKF filter, with the assumption of Gaussian noise, the remainder of this work will further investigate its operating characteristics. Chapter 11 will address the filter's ability to deal with different quality sensors, biases, and data dropouts.

11.0 Filter Behavior Under Other Than Nominal Conditions

The development thus far has considered the relative performance of four different algorithms given an "as expected" environment. The UKF-based filter using rate measurements was determined to exhibit the best overall performance. There are, however, a number of deviations from this "as expected" environment with which an attitude determination algorithm will have to contend. This chapter addresses several of these situations and evaluates the performance of the UKF-based filter using rate measurements under these "other than nominal" conditions.

11.1 Performance With Less Frequent Measurements

One important practical concern when implementing any filtering algorithm is how its performance varies with respect to the frequency at which measurements are available. This can have a significant impact on the number of sensors necessary to achieve the desired result and the processing power/speed required. The application of interest in this work, attitude determination for sounding rockets, assumes the use of Sun sensors, magnetometers, and rate gyroscopes. The magnetometers and the rate gyroscopes can theoretically be polled at any time, so measurement availability from these sensors does not drive the filtering cycle. The Sun sensor has a defined field of view, and measurements will only be available when the Sun is in this field of view. Since the vehicle is undergoing rotational motion, this field of view sweeps out a portion of the sky at each rotation. Obviously, how often Sun vector measurements are available is a function of how many Sun sensors are employed and what their field of view characteristics are. This measurement frequency is also clearly a function of the rotational motion of the vehicle. While the detailed design of an actual system requires many cost/benefits analyses to determine the number of sensors and their placement, a number of simulations were conducted to help characterize the performance of this filtering algorithm with respect to the availability of sensor measurements. This analysis assumes that only complete measurement sets are fed to the filter, meaning that Sun vector, magnetic field vector and body rate measurements are available at the filtering time step. Performance is analyzed for two data sets from the baseline example at various measurement intervals. The intervals are chosen, to a very rough approximation, to correspond to what would be available from Sun sensors given the 225 rev/min principle spin rate and one sensor, two sensors, three sensors, etc. Since these intervals assume the sensor only sees the Sun when it is directly on the bore sight of the sensor, these intervals are actually worse than what might be expected for an actual implementation. Again, the measurement interval is the result of many trade offs, and this analysis is meant to illustrate the trend with respect to performance versus interval. In each case, the parameters are tuned to the best possible figure of merit and Table 11.1 below is

Table 11.1: Performance at Varying Measurement Interval

Duration of Simulation (seconds)	Measurement Interval (sec)	Φ_S	$\tau_{\omega_1} = \tau_{\omega_2}$	τ_{ω_3}	Gauss-Newton Figure of Merit	UKF Figure of Merit
40 (set #2)	0.010	0.023	0.340	1e15	.0016	6.34e-4
40 (set #2)	0.020	0.030	0.240	1e12	.0016	8.57e-4
40 (set #2)	0.040	0.050	0.070	1e12	.0016	0.0011
40 (set #2)	0.070	0.070	0.060	1e13	.0016	0.0014
40 (set #2)	0.090	0.080	0.060	1e15	.0016	0.0015
40 (set #2)	0.135	0.340	0.050	7	.0016	0.0016
40 (set #2)	0.270	0.260	0.010	15	.0017	0.0017
40 (set #5)	0.005	0.018	0.430	1e12	.0016	4.49e-4
40 (set #5)	0.010	0.023	0.320	1e11	.0016	6.30e-4
40 (set #5)	0.020	0.031	0.190	1e12	.0016	8.48e-4
40 (set #5)	0.040	0.050	0.070	1e13	.0016	.0012
40 (set #5)	0.070	0.070	0.070	1e13	.0015	.0013
40 (set #5)	0.090	0.085	0.060	1e13	.0016	.0014
40 (set #5)	0.135	0.340	0.050	116	.0015	.0015
40 (set #5)	0.270	0.355	0.100	1e11	.0015	.0015

a summary of the results. Evaluation of the figure of merit results illustrates that for a given sensor accuracy, more frequent measurements yield a better solution. It is also evident, that there is a minimum time interval at which the UKF begins to provide a performance advantage over what is available from using only the Gauss-Newton portion of the algorithm. For these two sample data sets, this performance advantage is evident at measurement intervals of 0.09 seconds or less. While there is an improvement over the performance of Gauss-Newton alone, substantial improvement in performance is not seen until the measurement interval is closer to that predicted by the time step discussion in an earlier section. Given the motion simulated here, a sample rate, or filter time step, on the order of .01 sec is expected to generate good results, and that is the case in this data. This increase in the quality of the solution with more frequent measurements matches the intuitive assessment, as one would expect that more information would yield a better solution. In fact, as described in Chapter 7, this is not true in the general data fusion sense, since improperly combining additional information can sometimes lead to a worse solution than if the new information had not been included [61]. Fortunately, since the filter is designed to optimally estimate the solution, more information does indeed provide a better solution in this case. This type of assessment can be used by a system designer as one piece of information in his decision on how many sensors should be included.

The analysis in this section was accomplished by varying the measurement interval while holding the sensor accuracy constant. The next section provides examples of performance as the sensor accuracies are changed.

11.2 Performance With Sensors of Greater and Lesser Accuracy

The development of the filter algorithms and testing thus far has centered on baseline examples that use representative values for the sensor accuracies. These measurement standard deviations, 1.333 degrees for the Sun sensor, 3.333 degrees for the magnetometer, and 0.333 rev/min for the rate sensors, were determined in Chapter 6 as a best assessment of what is likely to be available to a low-cost system designer. In this section, representative performance of the algorithm given measurements of better, and worse, accuracy will be evaluated. To evaluate performance with the availability of better sensors, a Sun sensor with a measurement standard deviation of 0.5 degrees, a magnetometer with a measurement standard deviation of 1.0 degrees and rate sensors with measurement standard deviations of 0.25 rev/min were simulated. For the case of worse sensors, measurement standard deviations of 5 degrees, 10 degrees and 1 rev/min were used. The simulated profile matches that of the baseline example, with an asymptotic approach to final rates of 0.5 rev/min about the body "x" axis, 0.5 rev/min about the body "y" axis, and 225 rev/min about the body "z" axis. For each case, the filter parameters are tuned for the optimum overall figure of merit, as has been done consistently throughout this work. Tables 11.2 and 11.3 summarize the results from these simulations. While the figure of merit is the adopted overall performance measure, the mean values for the "pointing accuracy" angles corresponding to each of the body axes are also given to help visualize the relative performance as a function of sensor accuracy. As expected, the better the sensor, the better the solution. What is of note, however, is that the filter provides the greatest improvement for scenarios using less accurate sensors. The improvement in figure of merit for the first case above is on the order of 61 percent. For the most accurate sensor case, it is approximately 58 percent. For the least accurate sensors simulated above, the improvement is on the order of a huge 76.7 percent. So, while the filter works well with any of these sensors, it appears to be especially well-suited to lower performing sensors, like those envisioned for this application. Having examined the filter behavior with respect to changes in measurement interval and changes in sensor accuracy, the next section investigates the impact of the loss of measurements on the algorithm.

Table 11.2: Sample Filter Performance for Sensors of Varying Accuracy

Simulation	Φ_S	$\tau_{\omega_1} = \tau_{\omega_2}$	τ_{ω_3}	Gauss-Newton Figure of Merit	UKF Figure of Merit
40 (asymptotic, 1.333°, 3.333°, 0.333 rpm)	0.023	0.32	1e11	0.0016	6.30e-4
40 (asymptotic, 0.5°, 1.0°, 0.25 rpm)	0.022	0.24	1e11	6.91e-4	2.93e-4
40 (asymptotic, 5°, 10°, 1.0 rpm)	0.046	0.17	1e13	0.0146	0.0034

Table 11.3: Error Angle Comparison as a Function of Sensor Accuracy

Simulation	Φ_S	$\tau_{\omega_1} = \tau_{\omega_2}$	τ_{ω_3}	GN Mean Error Angle 1 (deg)	GN Mean Error Angle 2 (deg)	GN Mean Error Angle 3 (deg)	UKF Mean Error Angle 1 (deg)	UKF Mean Error Angle 2 (deg)	UKF Mean Error Angle 3 (deg)
40 (asymptotic, 1.333°, 3.333°, 0.333 rpm)	0.023	0.32	1e11	2.81	2.80	3.33	1.82	1.85	2.06
40 (asymptotic, 0.5°, 1.0°, 0.25 rpm)	0.022	0.24	1e11	0.93	0.92	1.04	0.88	0.87	0.97
40 (asymptotic, 5°, 10°, 1.0 rpm)	0.046	0.17	1e13	8.86	8.92	9.99	4.47	4.53	4.58

11.3 Behavior in the Event of a Loss of Measurements

Another situation that may occur is the loss of measurements for some period of time due to telemetry problems, on-board hardware problems, or a host of other reasons. While total loss of data is an unrecoverable situation, lost measurements or short periods without data may occur. To be useful, in this event, the filter algorithm should be well-behaved and recover gracefully to provide acceptable estimates. This section evaluates the performance of the UKF algorithm under these conditions. Six different approaches were investigated for dealing with a loss of measurements. Approach 1 consists of simply setting the Gauss-Newton produced quaternion back to [0 0 0 1] and letting the rates be processed as being equal to zero for each time step where measurements are missing. Approach 2 involves setting the quaternion "measurement" equal to the previous "measurement" and leaving the rotational rates equal to zero at time steps that do not have a measurement. Approach 3 sets the quaternion "measurement" to the last "measurement," but now sets the rate measurements to the last rate measurement for each time step that has a measurement missing. The fourth approach implements approach 3, but also makes a change in the filter code. The "measurement" vector is set equal to the predicted observation vector, in essence making the filter residual equal to zero, thereby weighting the measurement at zero and making the next state estimate equal to the projected state. Approach 5 sets the Gauss-Newton quaternion "measurement" equal to the last quaternion estimate out of the filter and the rate measurements equal to the last rate estimates out of the filter. Finally, approach 6 implements approach 5, but also sets the "measurement" vector in the filter equal to the predicted observation vector. Although more sophisticated methods may certainly be devised, each of the approaches described here allow the filter to successfully continue estimating the state vector despite a measurement loss. Each was tested against the baseline example with a one second loss of data in the center of the "knee" in the rate profiles. The data "drop out" begins at 9.5 seconds and continues to 10.5 seconds. Table 11.4 summarizes the results. Clearly methods 5 and 6 are

Table 11.4: Sample Performance With One Second Measurement Loss

Design Approach	Duration of Simulation (seconds)	Φ_S	$\tau_{\omega_1} = \tau_{\omega_2}$	τ_{ω_3}	Gauss-Newton Figure of Merit (With Dropout)	UKF Figure of Merit (No Dropout)	UKF Figure of Merit (With Dropout)
Approach 1	40 (set #2)	0.023	0.360	1e7	3.94	6.34e-4	0.4874
Approach 2	40 (set #2)	0.023	0.360	1e7	3.94	6.34e-4	3.850
Approach 3	40 (set #2)	0.023	0.360	1e7	0.0110	6.34e-4	0.0102
Approach 4	40 (set #2)	0.023	0.360	1e7	0.0110	6.34e-4	0.0014
Approach 5	40 (set #2)	0.023	0.360	1e7	0.0017	6.34e-4	8.25e-4
Approach 6	40 (set #2)	0.023	0.360	1e7	0.0017	6.34e-4	8.25e-4

significantly better methods, with performance approaching that of the case with no dropouts. A one second dropout is not likely, but serves to illustrate that the algorithm can be made robust with respect to loss of measurements. Figures 11.1 through 11.10 illustrate the filter estimates of each state before, during and after the one second dropout from 9.5 to 10.5 seconds. While there is a slight degradation in performance, as determined by the figures of merit, relative to the run without a dropout, the filter does not diverge or fail to operate and continues to provide very good estimates.

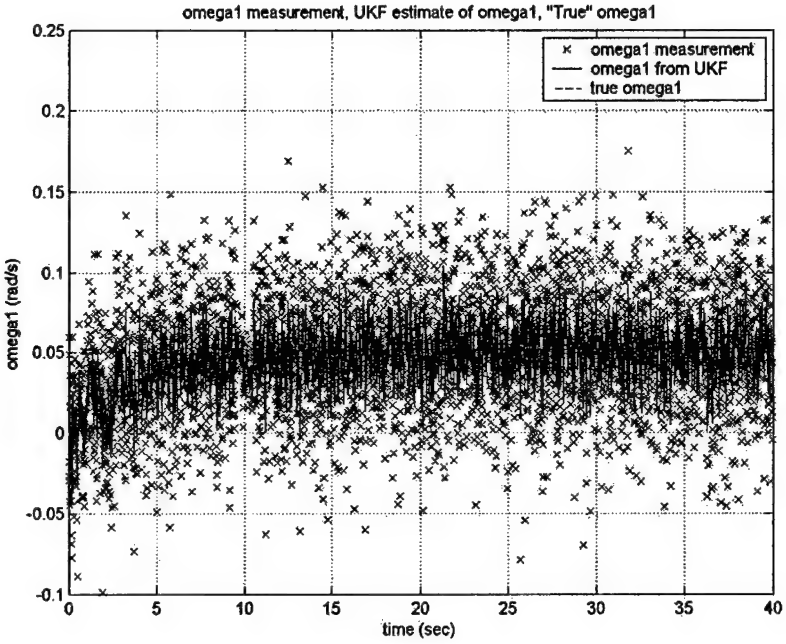


Figure 11.1: UKF Estimate of ω_1 Through One Second Measurement Dropout

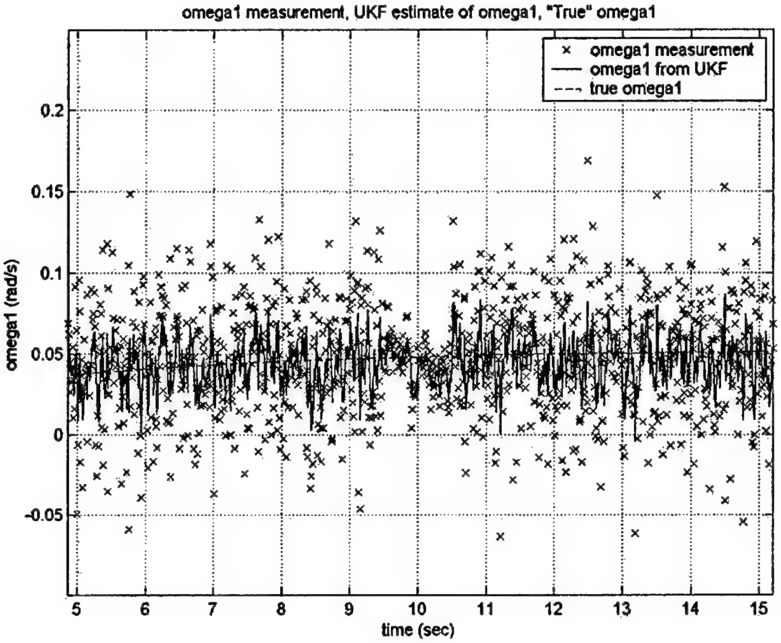


Figure 11.2: Close-up of UKF Estimate of ω_1 Through One Second Measurement Dropout

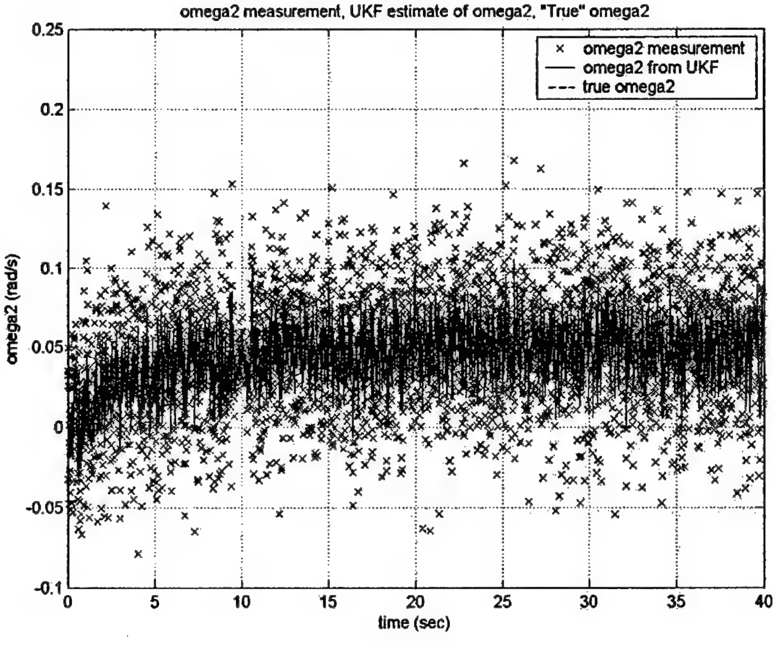


Figure 11.3: UKF Estimate of ω_2 Through One Second Measurement Dropout

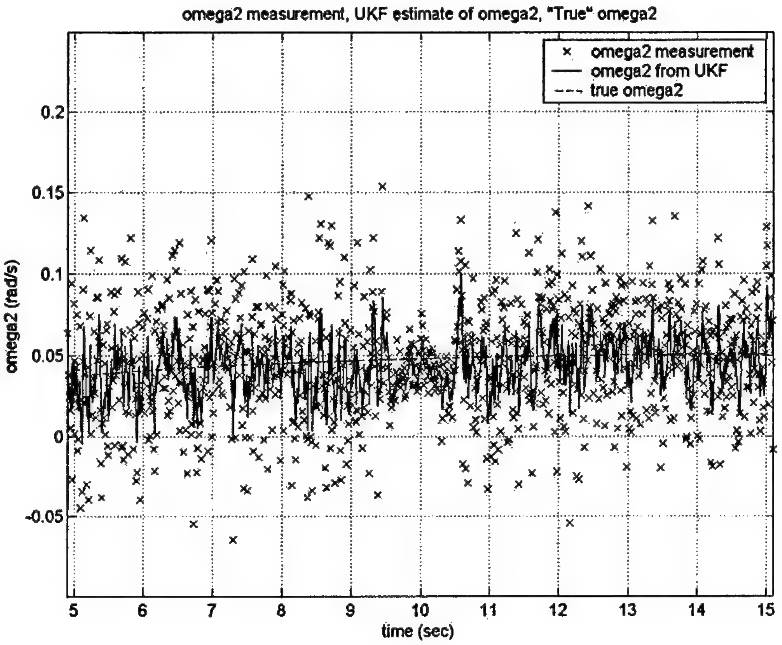


Figure 11.4: Close-up of UKF Estimate of ω_2 Through One Second Measurement Dropout

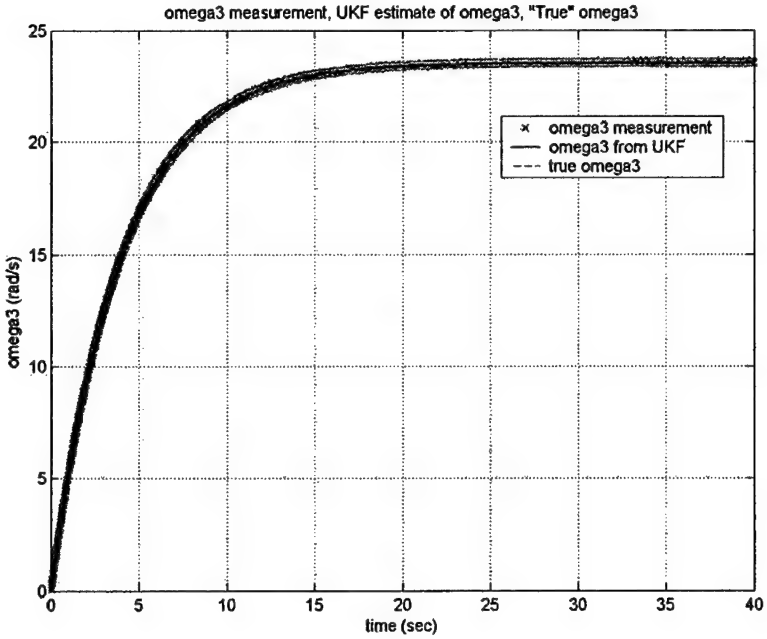


Figure 11.5: UKF Estimate of ω_3 Through One Second Measurement Dropout

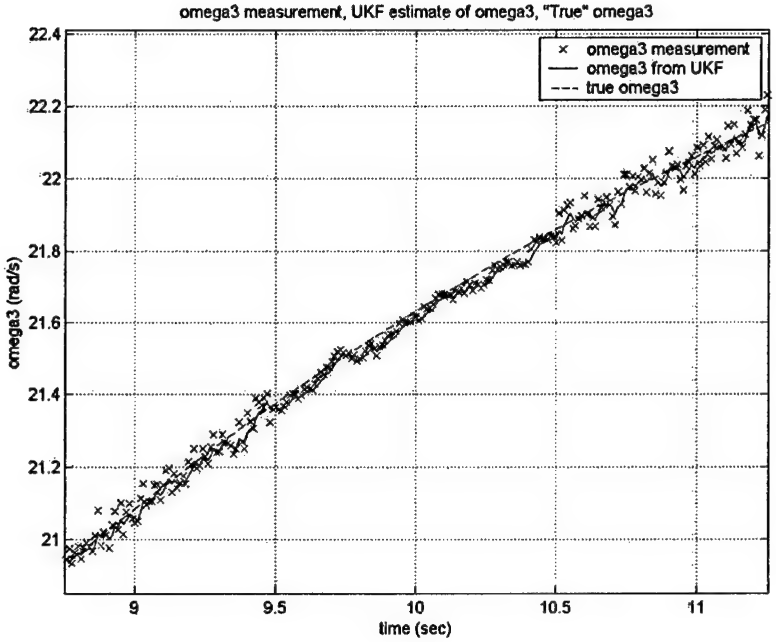


Figure 11.6: Close-up of UKF Estimate of ω_3 Through One Second Measurement Dropout

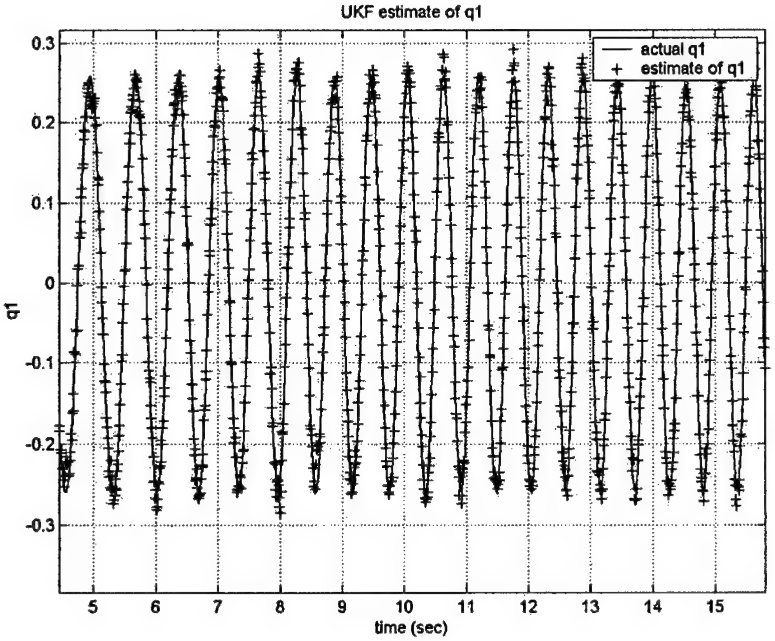


Figure 11.7: Close-up of UKF Estimate of q_1 Through One Second Measurement Dropout

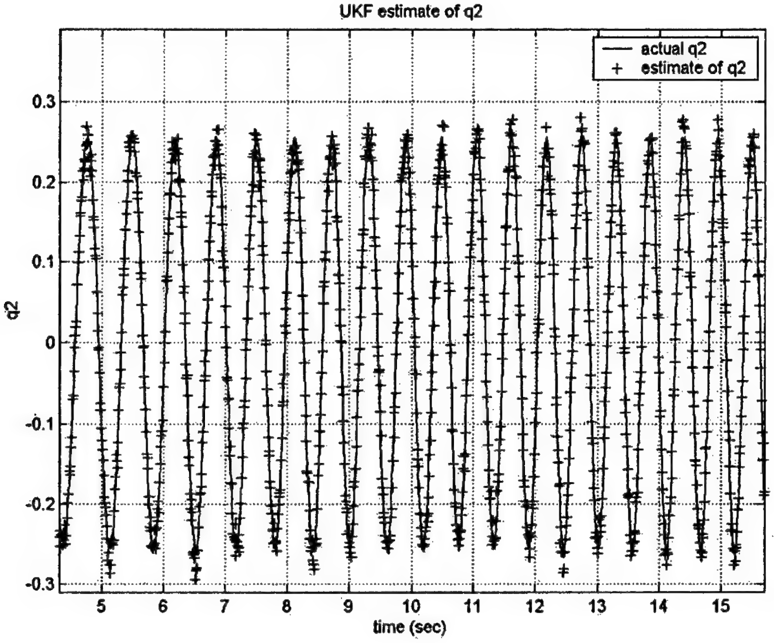


Figure 11.8: Close-up of UKF Estimate of q_2 Through One Second Measurement Dropout

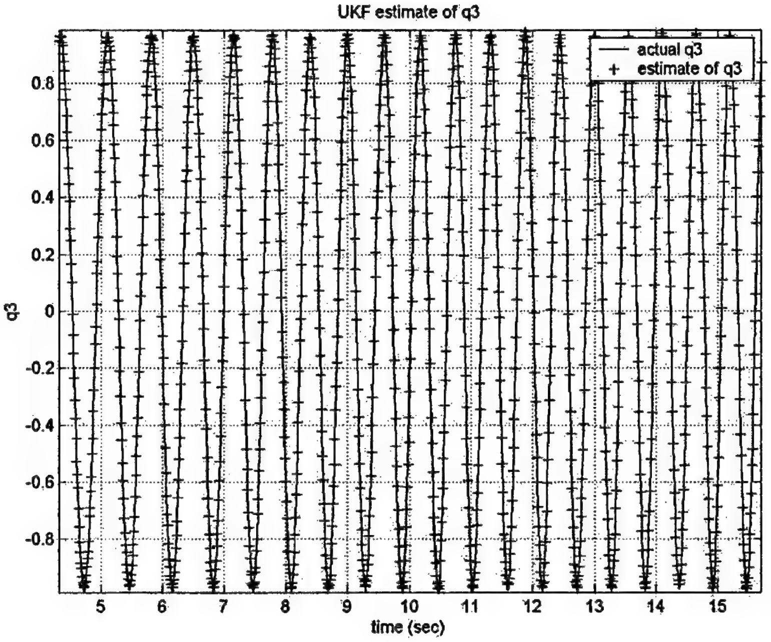


Figure 11.9: Close-up of UKF Estimate of q_3 Through One Second Measurement Dropout

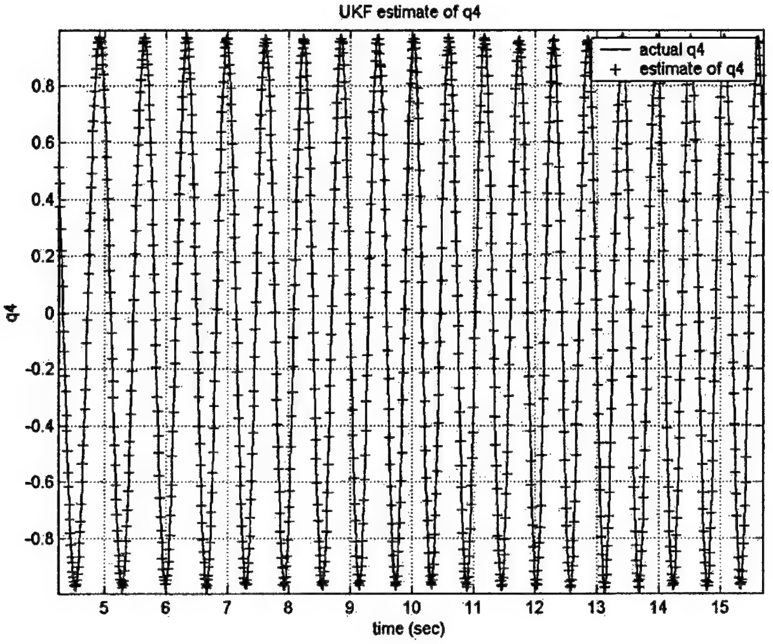


Figure 11.10: Close-up of UKF Estimate of q_4 Through One Second Measurement Dropout

11.4 Impact of a Sensor Bias

One of the more insidious situations that may be encountered is a bias on one or more of the measurements. When this occurs, all indications may be that the filter is operating properly, when in fact the solution is being corrupted at each step. While the filter does not diverge or cease to function, the quality of the solution may be in question. Unfortunately there is no straightforward solution to such a situation. Especially in the case where there are no redundant sensors, there will be no real way to tell if the filter is providing realistic, but incorrect answers. One possible mitigating approach is to have several sensors that may be compared. In this case, if there are at least three, a majority voting scheme might be employed to determine if one sensor has a bias or other problem. If such a sensor could be identified, its input to the algorithm could be eliminated. This might be possible, for instance, if three Sun sensors were employed to allow for more frequent measurements, as discussed in Section 11.1. If one of these sensors was identified as problematic, its input might be disregarded, with a resulting gain in performance due to eliminating the bias. This performance gain, however, could possibly be offset by a negative impact due to an increased measurement interval. The net effect on performance would depend on the particular system under consideration. Unfortunately, the inclusion of redundant sensors poses a number of problems with respect to the constraints on a low-cost system approach. To evaluate the impact on performance of the UKF filter when a bias is present on a sensor, three situations were evaluated. First, one component of the Sun vector measurement was given a constant bias. Second, a constant bias was imposed on one component of the magnetic field vector measurement. Finally, one rate sensor had a constant bias added to its measurement. Again, the baseline 40 second example was utilized for comparison purposes. Table 11.5 summarizes the results of this testing

Table 11.5: UKF Filter Performance With Sensor Bias

Duration of Simulation (seconds)	Bias Imposed on Simulated Measurement	Φ_S	$\tau_{\omega_1} = \tau_{\omega_2}$	τ_{ω_3}	Gauss-Newton Figure of Merit (No Bias)	Gauss-Newton Figure of Merit (With Bias)	UKF Figure of Merit (No Bias)	UKF Figure of Merit (With Bias)
40 (set #2)	+0.1 on "x" component of Sun vector measurement	0.023	0.36	1e7	.0016	.0021	6.34e-4	.0013
40 (set #2)	+0.1 on "y" component of mag field vector measurement	0.023	0.36	1e7	.0016	.0023	6.34e-4	.0015
40 (set #2)	+0.1 rev/min on "x" axis rate measurement	0.023	0.36	1e7	.0016	.0047	6.34e-4	.0034

Another possible bias to be aware of, which was not simulated, would be introduced by errors in the reference models. For example, this would occur if the magnetic field vector was perturbed by electrical currents within the rocket. This could potentially cause a bias on all three of the magnetometer measurement axes. From the results in Table 11.5, it is clear that a constant bias is a troubling development. In these cases, the filter continues to successfully estimate the state vector, albeit with greater error. There is no divergence or other tell-tale indication of a problem. From a positive perspective, the filter continues to operate normally and there is no catastrophic result. Depending upon the bias, the impact may be great or small. The results from the third trial above indicate poor overall results, but in fact, this bias on the rate sensor has a small impact on the estimation of the quaternion. Despite the accurate attitude determination, the poor rate estimate figures into the overall performance measure and the overall figure of merit suffers. The first two trials were for biases on attitude sensors, and these translate directly into performance impacts on the attitude determination. Short of adding redundant sensors, there is no obvious solution to this problem. Therefore, significant emphasis should be placed on testing the sensors before launch, and characterizing any measurement biases that may be present.

11.5 Performance When Actual Motion is Different than Predicted Motion

In addition to loss of measurements, biases, and the issues surrounding the accuracy of the sensors, the motion encountered by the rocket may be very different from the "nominal" motion expected and for which the filter parameters were tuned. It is important to understand what the filter performance is like in such a situation. Fortunately, the error minimization portion of this algorithm makes it very robust to changes in the character of the motion, with respect to determining the attitude. Extensive simulation has demonstrated that the filter is able to accurately estimate the orientation of the rocket even for motion radically different than that for which the filter parameters were tuned. In these cases, what suffer significantly are the estimates of the body rates. As an example, simulations were conducted using 40 second runs with the default parameters and the optimum process noise and time constants from the baseline example. The rate profiles were changed from rates that asymptotically approached 0.5 rev/min about the body "x" axis, 0.5 rev/min about the body "y" axis and 225 rev/min about the body "z" axis to profiles that simulate significantly different motion. In this manner, the filter performance is evaluated when the parameters are tuned for one expected motion profile, but the filter encounters something different. The first trial uses the same final rotation rates as the baseline example, but in this case they are fixed, instead of asymptotically approaching the final values. The second trial simulates motion with fixed rates of 30 rev/min about each body axis. This translates into motion represented by a rate of 59.96 rev/min about the (1,1,1) axis in the body frame. The third example presented here corresponds to a change of principle spin axis from the "z" axis to the "x" axis, with fixed rates of 30 rev/min about the body "x" axis, 0 rev/min about the body "y" axis, and 5 rev/min about the body "z" axis. Trial 4 again has rates of 30 rev/min about each axis, but now these are approached asymptotically as in the baseline example. Finally, trial 5 is the asymptotic case using rates of 30 rev/min about the body "x" axis, 0 rev/min about the body "y" axis, and 5 rev/min about the body "z" axis. These motions vary significantly from that of the baseline example, and the results of the simulation and filtering are summarized in Table 11.6 below. Since a different random noise component, but one having the same standard deviation, is added during each simulation, the data sets used in these cases are not identical to the 40 second baseline

Table 11.6: UKF Filter Performance During Unpredicted Motion

40 Second Simulation	Gauss-Newton ω Performance	Gauss-Newton MSE for Rotated Vectors	Gauss-Newton Figure of Merit	UKF ω Performance	UKF MSE for Rotated Vectors	UKF Figure of Merit	UKF Figure of Merit (nominal asymptotic)
Trial 1 (fixed ω_x=0.5 rpm ω_y=0.5 rpm ω_z=225 rpm)	0.0012	9.92e-4	0.0016	2.43e-4	4.39e-4	5.02e-4	6.34e-4
Trial 2 (fixed ω_x=30 rpm ω_y=30 rpm ω_z=30 rpm)	0.0012	0.0010	0.0016	0.0189	4.60e-4	.0189	6.34e-4
Trial 3 (fixed ω_x=30 rpm ω_y=0 rpm ω_z=5 rpm)	0.0012	0.0010	0.0016	0.0098	4.52e-4	0.0098	6.34e-4
Trial 4 (asymptotic ω_x=30 rpm ω_y=30 rpm ω_z=30 rpm)	0.0012	9.77e-4	0.0016	0.0159	9.77e-4	0.0159	6.34e-4
Trial 5 (asymptotic ω_x=30 rpm ω_y=0 rpm ω_z=5 rpm)	0.0012	9.92e-4	0.0016	0.0081	4.44e-4	0.0081	6.34e-4

example. However, the values for the "nominal" motion are typical of the baseline example, and adequate for a qualitative comparison of performance. As noted earlier, and made clear by these results, the orientation of the rocket is always estimated at close to the same accuracy as achieved when processing the expected motion. For the case of fixed rates with the same final values as the baseline case, the filter performance is even better for the fixed rate case than for the asymptotic case, due to the ease of estimating the simpler, non-accelerating motion. In cases where the motion is characterized by different final rates, however, the overall figure of merit is severely degraded, and this is driven by the much less accurate estimation of the body rates. One possible fix, if such degradation in performance can be detected, is to increase the process noise or the time constants used by the filter. While the finely tuned parameters that are optimized for the expected motion produce the best results under ideal conditions, an actual implementation might warrant investigating the use of less-than-optimum values in order to achieve greater robustness in the face of unexpected events. The next section examines the possibility of detecting such an occurrence, what might be done to compensate for it, and applies such an anomaly detection method to one of the trials investigated here.

11.6 Determining if the Filter is Operating Correctly

In a simulation scenario when the truth is readily available, it is straightforward to compare the results coming out of the filter to the truth and assess performance of the algorithm. When evaluating filter error in a simulated environment, the error in the estimate of each state is expected to fall within the theoretical bounds at least 68 percent of the time [17]. In effect, we are using the expectation that in a Gaussian distribution we expect 68 percent to fall within 1σ, or one standard deviation, of the mean. For a Kalman filter-based algorithm, this theoretical boundary is found in the covariance matrix, *P*, which is calculated as part of the iterative process and is available at each time step. *P* is the expected value of the difference between the actual state vector and the estimate of the state vector times the transpose of this quantity. The mathematical representation is

$$P = E[(x-\hat{x})(x-\hat{x})^T]. \tag{11.1}$$

Since the variance of a random variable can be expressed as [17]

$$\sigma^2 = E\left(\left[x - E(x)\right]^2\right) = E(x^2) - E^2(x) \tag{11.2}$$

the elements of the diagonal covariance matrix may be treated as the variances of the error in the estimates of the corresponding state vector elements. Taking the square root of the diagonal elements, then, yields the predicted standard deviations of the error in the estimates of the states. It is this "+/- 1σ value" at each time step that forms the theoretical boundary of the error in the estimate that is seen in many of the figures. Once the algorithm is implemented, however, the "truth" is not readily available. Since the filter may be operating in what appears to be a nominal fashion, but may be producing incorrect results, it would be very helpful to have a flag, as such, that would indicate a problem with the algorithm. Such a flag could then be used as a trigger to modify the parameters in the algorithm or to resort to a secondary method that may not be as accurate, but that is not subject to whatever is causing problems with the primary algorithm.

One such "flag" that is sometimes used for Kalman filters operating in real world applications is an evaluation of the "residual". The residual is the quantity that is multiplied by the Kalman gain when determining the state estimate. As seen in equation (9.46) for the EKF and again in equation (9.73) for the UKF, the residual is the difference between the projected observations and the actual observations based on sensor measurements. Unlike the true value of the states, both of these quantities are available during real world operation. As Zarchan points out

> If a theoretical prediction of the residual can be derived, we can say that the filter is working properly if the measured residual falls within the theoretical bounds 68% of the time. If the residual continually exceeds the theoretical bounds by a large factor (i.e., three), we can conclude that the filter is diverging and that some action is required to prevent complete catastrophe (i.e., add more process noise or switch to higher-order filter). If the residual instantly exceeds the theoretical threshold bounds by a large value, the data might be bad, and we might want to ignore that measurement [17].

Development and successful operation of this form of "anomaly detection" is demonstrated in [17] as an effective means for detecting divergence in a polynomial Kalman filter. The development from [17] begins with the linear Kalman filter equation for the state estimate

$$\hat{x}_k = \Phi_k \hat{x}_{k-1} + K_k (z_k - H\Phi_k \hat{x}_{k-1}) \tag{11.3}$$

where Φ, $\hat{x}$, K, and H are as defined in Chapter 8. It should be noted that, unlike the linear Kalman filter for which this development takes place, the EKF and UKF do not use the state transition matrix, Φ, to propagate states forward. It does appear in the EKF formulation as seen in Chapter 8, but is used for determination of the various matrices leading up to calculating the Kalman gain, not for state propagation. The UKF formulation uses a different approach for determination of these matrices and the Kalman gain that does not rely on Φ. From (11.3), it is evident that the residual is equal to

$$\mathit{residual} = z_k - H\Phi_k \hat{x}_{k-1}. \tag{11.4}$$

For the case where the system has a linear measurement equation, the observation z_k is found using

$$z_k = Hx_k + v_k \tag{11.5}$$

where the next state, in the linear case, is obtained from

$$x_k = \Phi_k \hat{x}_{k-1} + w_k. \tag{11.6}$$

Substituting (11.4) and (11.5) into (11.6) yields an expression for the residual of

$$residual = H\Phi_k x_{k-1} + Hw_k + v_k - H\Phi_k \hat{x}_{k-1}. \qquad (11.7)$$

Combining like terms produces

$$residual = H\Phi_k (x_{k-1} - \hat{x}_{k-1}) + Hw_k + v_k. \qquad (11.8)$$

Using the definition of a covariance, the covariance of the residual is found by multiplying (11.8) by its transpose and taking the expectation of both sides. Zarchan points out the necessary assumptions that "the process and measurement noise are not correlated and that neither noise is correlated with the state or its estimate [17]." Given these assumptions, the covariance of the residual is found to be

$$E(residual * residual^T) = H\Phi_k E\left[(x_{k-1} - \hat{x}_{k-1})(x_{k-1} - \hat{x}_{k-1})^T\right]\Phi_k^T H^T$$

$$+ HE(w_k w_k^T)H^T + E(v_k v_k^T). \qquad (11.9)$$

Because

$$P_k = E\left[(x_{k-1} - \hat{x}_{k-1})(x_{k-1} - \hat{x}_{k-1})^T\right] \qquad (11.10)$$

$$Q_k = E(w_k w_k^T) \qquad (11.11)$$

$$R_k = E(v_k v_k^T) \qquad (11.12)$$

the covariance of the residual can be reduced initially to

$$E(residual * residual^T) = H\Phi_k P\Phi_k^T H^T + HQ_k H^T + R_k \qquad (11.13)$$

and then eventually to

$$E(residual * residual^T) = H(\Phi_k P\Phi_k^T + Q_k)H^T + R_k. \qquad (11.14)$$

As defined in Chapter 8, the covariance matrix before the update is equal to

$$M_k = \Phi_k P_k \Phi_k^T + Q_k. \qquad (11.15)$$

Substituting (11.15) into (11.14) yields

$$E(residual * residual)^T = HM_k H^T + R_k \qquad (11.16)$$

where H is the measurement matrix, M is the covariance matrix before the update, and R is the measurement noise matrix as defined in Chapter 8. While this development is based on the linear Kalman filter structure, each of the matrices in the final expression is also available in the UKF formulation. Especially for this application where, despite the nonlinearities in the problem, the measurement matrix H is essentially the identity matrix, it is hopeful that (11.16) should provide similar warning of divergence, or other serious breakdown, in the operation of the filter. To determine if this is true, simulations were run in which the filter is obviously working properly and others where it is not. The following two figures show plots from one of these simulations of the error in the state estimate and of the "residual test" for one state from the nominal case. Here it is clear that both the error in the state estimate, which is available only in simulation when the true value of the state is known, and the residuals corresponding to this state fall within the theoretical bounds at least 68 percent of the time. This indicates nominal operation of the filter. Figures 11.13 and 11.14 are for the same state, when the filter is not operating properly. This anomalous operation is generated by using the same baseline simulation as for the earlier figures, but extending the measurement interval to 0.1 seconds versus 0.01 seconds without a corresponding retuning of the filter process noise. The first of the two shows that the error in the state estimate does not satisfy the 68 percent criteria, indicating a problem. Figure 11.14 demonstrates that the filter residuals also violate the 68 percent rule with respect to their theoretical boundaries, thereby also indicating a problem has occurred. Note that the residuals begin to violate the theoretical bounds at approximately the same time in the run as the state estimate error begins to deviate. Since the elements needed to produce the second plot are available without apriori knowledge of the true state value, this suggests a promising approach to evaluating the UKF filter for proper operation and serving as a "flag" in real world operation. Unfortunately, in cases where the filter is not operating at its best, but is not experiencing catastrophic failure, the residual test may not be a definitive indicator for whether or not to switch filter parameters or

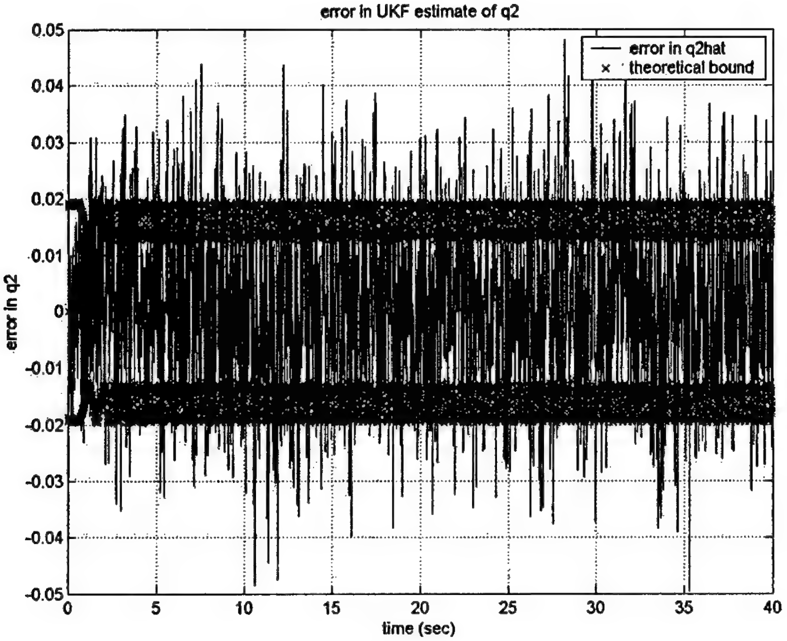


Figure 11.11: Sample Error in State Estimate When Filter is Working Properly

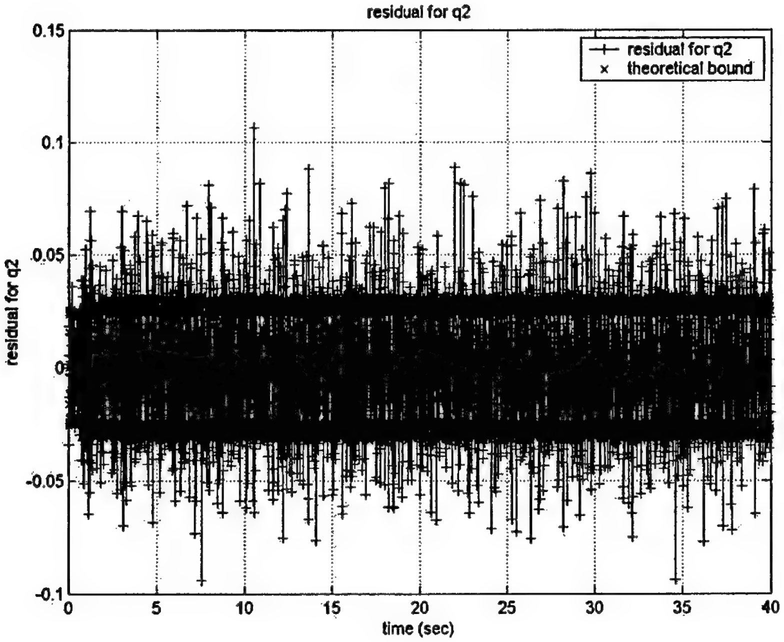


Figure 11.12: Sample Filter Residual for One State When Filter is Working Properly

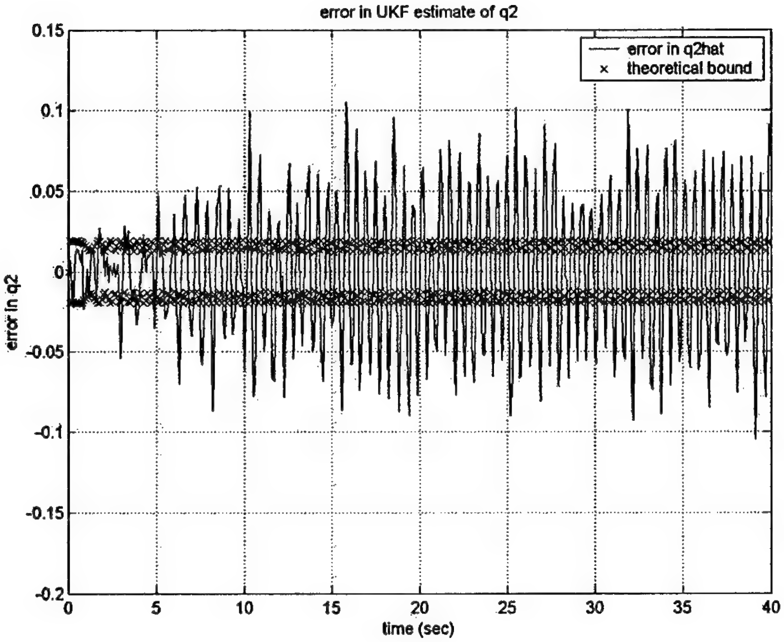


Figure 11.13: Sample Error in State Estimate When Filter is Working Improperly

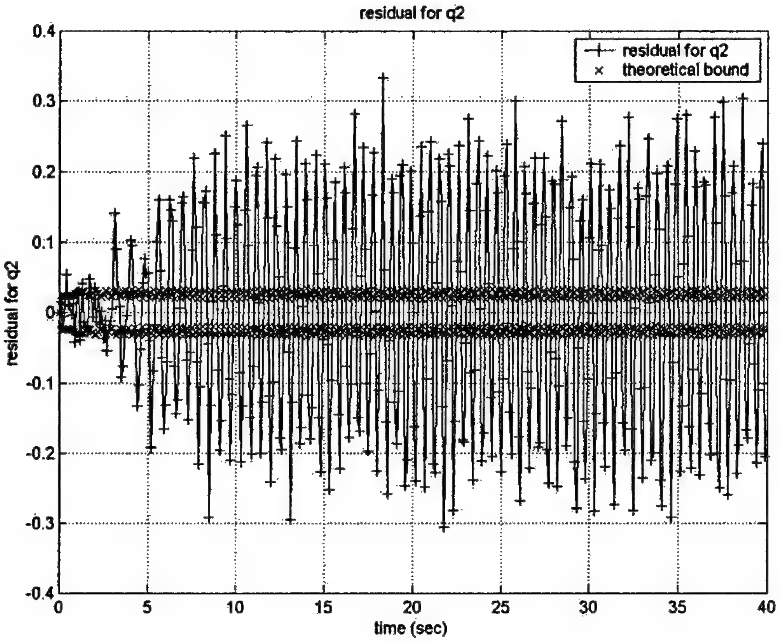


Figure 11.14: Sample Filter Residual for One State When Filter is Working Improperly

methods. The bias problem discussed in an earlier section might be one example. Based on the parameters of a particular problem, the system designer would need to establish threshold values to serve as triggers for making such a determination. To illustrate what this type of anomalous filter performance means in terms of attitude determination, the following two figures show plots of the error angle for the first body frame axis when the filter is working as expected, and when performing anomalously. As expected, the error angle increases rapidly once the filter estimates begin to diverge. The plots in this section illustrate the performance for only one of the states and the error for only one body axis. When implementing an error detection method, such as that described in this section, the residuals corresponding to each state should be evaluated since the filter may continue to adequately estimate certain states while failing to correctly estimate others. Another example where this method certainly works is the case where the motion is other than expected, as discussed in the previous section. As an illustration, the data set from trial 3 of the previous section where the motion is a fixed 30 rev/min about the body "x" axis, 0 rev/min about the body "y" axis and 5 rev/min about the body "z" axis was re-evaluated with only a change to the $\tau_1 = \tau_2$ parameter. As shown in Figure 11.17, the residual for the estimate of ω_1, when using the optimized parameter, clearly violates the theoretical boundary established from the covariance of the residual.

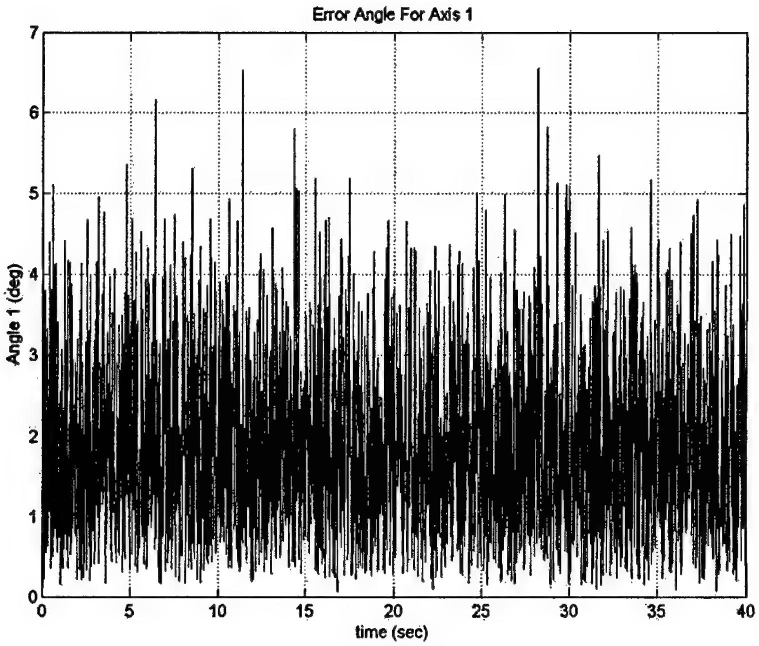


Figure 11.15: Sample Error Angle Performance When Filter is Operating Properly

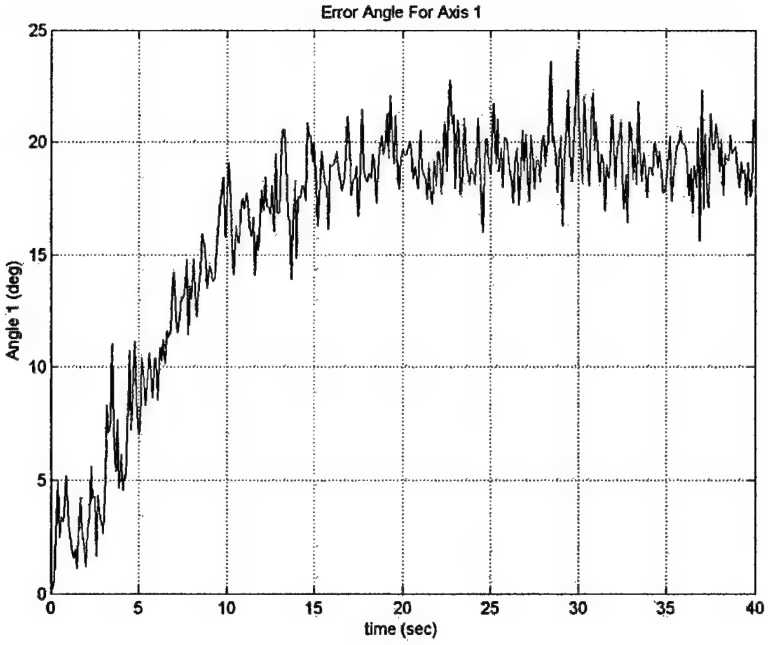


Figure 11.16: Sample Error Angle Performance When the Filter is Operating Improperly

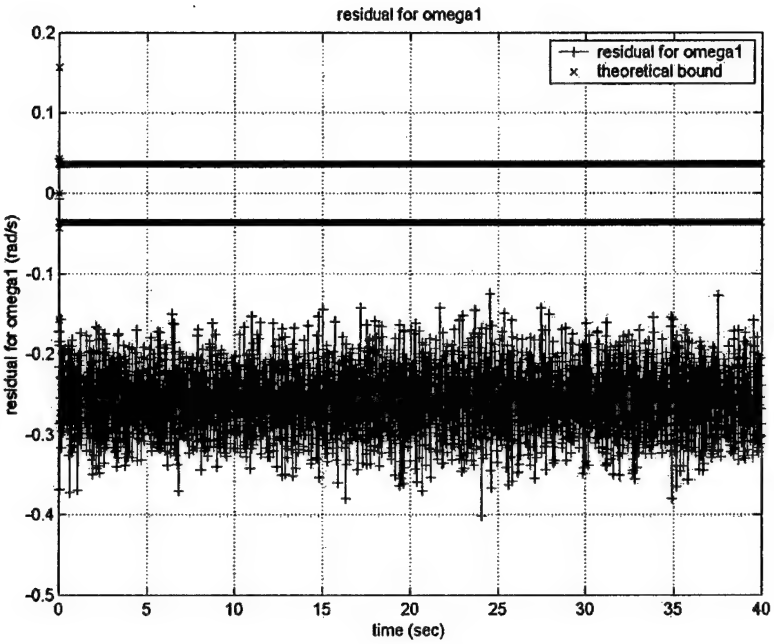


Figure 11.17: Example of Clear Residual Departure Indicating Improper Filter Performance

Figure 11.18 illustrates the corresponding filter estimate of ω_1. If, in a real application, this was observed, the time constants and/or process noise could be adjusted. For instance, changing τ_1 and τ_2 to equal 1e7 versus the "optimized" 0.34 yields approximately the same attitude determination performance, but radically improves the estimation of the rotational rates and overall figure of merit. Figure 11.19 illustrates the new residual for ω_1 which now satisfies the 68 percent criteria relative to the theoretical bounds and indicates proper filter performance. Figure 11.20 demonstrates the much improved estimation of ω_1 and Table 11.7 summarizes the change in performance associated with this change in parameter. From an implementation standpoint, either the "optimized" filter parameters could be set to more conservative values to increase filter robustness, with a corresponding decrease in performance under ideal conditions, or the algorithm could be designed to observe the residual and make changes accordingly. Examples where this method appears to work, like the two illustrated here, may certainly be found. While this method holds promise as a "flag" for improper filter performance, the indicators are not necessarily as clear as those observed for the linear Kalman filter case demonstrated in [17], and further research is warranted regarding the reliability of this approach for the UKF-based filter.

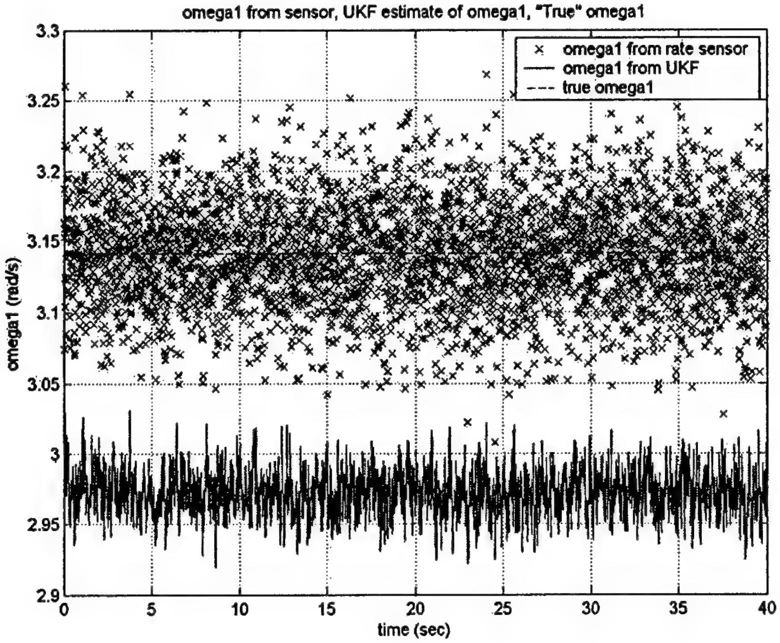


Figure 11.18: Filter Estimate of ω_1 Corresponding to Improper Filter Performance

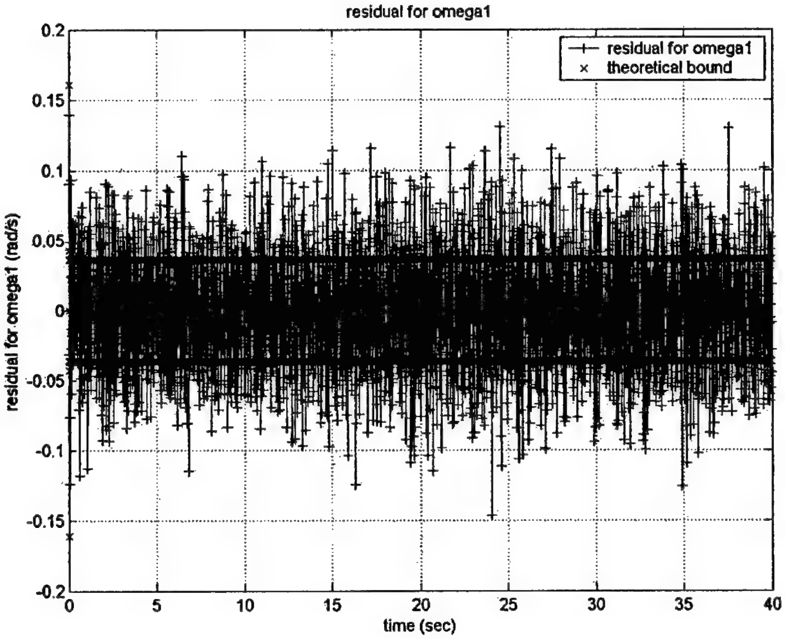


Figure 11.19: Residual Performance Following Change in Time Constant

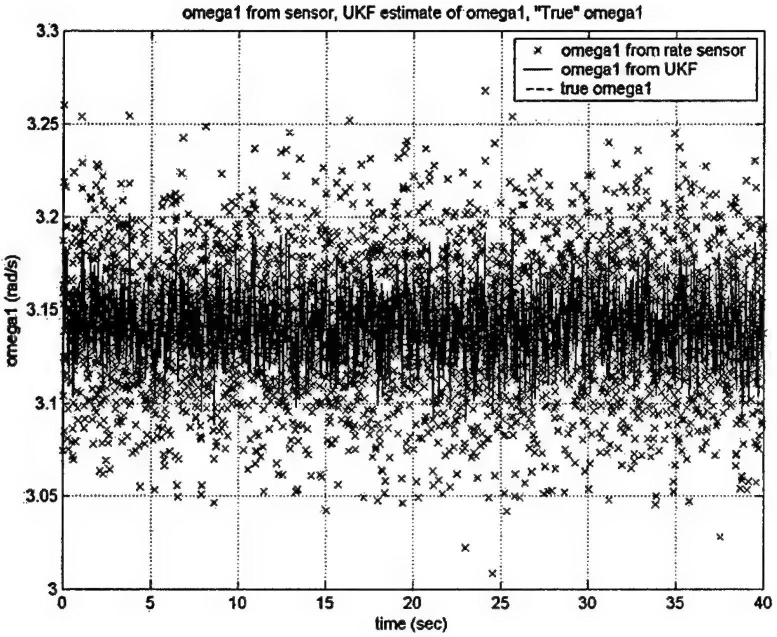


Figure 11.20: Filter Estimation of ω_1 Following Change of Time Constant

Table 11.7: Summary of Performance Before and After Change of Time Constant

Trial 3 from Table 11.6 (fixed ω_x=30 rpm ω_y=0 rpm ω_z=5 rpm)	Gauss-Newton ω Performance	Gauss-Newton MSE for Rotated Vectors	Gauss-Newton Figure of Merit	UKF ω Performance	UKF MSE for Rotated Vectors	UKF Figure of Merit
$\tau_1 = \tau_2 = 0.34$	0.0012	0.0010	0.0016	0.0098	4.52e-4	0.0098
$\tau_1 = \tau_2 = 1e7$	0.0012	0.0010	0.0016	2.53e-4	4.50e-4	5.16e-4

11.7 A Discussion of Error in the Estimate

Having developed the algorithm and evaluated its performance, it is important to identify error sources that may impact the accuracy of the solution. One not very obvious limitation on this method is the accuracy of the solar ephemeris and magnetic field models from which the reference vectors are computed to be fed into the Gauss-Newton error minimization routine. The development herein assumes the availability of these reference vectors at each time step. While the solar ephemeris is typically very well known, the magnetic field models are not nearly as accurate and are constantly being refined. Additionally, while the Sun is a relatively constant parameter, the magnetic field is continuously changing, primarily due to solar activity. To gain maximum accuracy out of the UKF-based approach, the most accurate model values possible must be used. To pull reference vectors from either model requires the location of a point of interest. Again, in this development, the position of the rocket is assumed to be known, and would most likely come from either radar or GPS measurements in a real-world application. As noted in Chapter 1, great progress is being made toward reliable implementation of GPS positioning on sounding rockets, and this will help minimize the errors in the position vector that would be used to calculate solar and magnetic field reference vectors from the appropriate models. Nevertheless, some level of error will be associated with the modeled values and this will translate into error in the attitude solution. This error is not unique to this attitude determination method, but is one that a system designer must take into account. Obviously the errors in the sensor measurements contribute to the overall error in the solution, and Chapters 10 and 11 have attempted to give a characterization of the amount of error that would be typical for a given sensor accuracy. Numerical processing errors such as those due to round off, integration step size and method, and iteration tolerance contribute to the overall error. As discussed in the sections on algorithm design and the tuning of parameters, attempts have been made to minimize these errors while taking into account the desire to produce an algorithm that can be implemented in real time. The accuracy of the estimates produced is also impacted by how close the actual motion is to the motion prototype used to “tune” the filter. The farther the true motion is from the predicted motion, the less accurate the estimate. Finally, as noted in [39], there are random errors at each stage of the procedure.

12.0 Conclusions

12.1 Significant Conclusions

The principle result of this work is a new algorithm for the attitude determination of a rotating body using low-cost sensors. This algorithm is based on a quaternion formulation to avoid the well-known singularities associated with Euler angles and other three-parameter attitude representations. In all, four algorithms were designed, programmed and tested. Each used a Gauss-Newton error minimization to reduce the number of measurements to be processed by the filter by reducing two vector measurements into a single four-element quaternion. These four elements, combined with three body rates, yield seven measurements to be processed by the filter instead of the original nine. The sensors assumed for this development are a Sun sensor, providing a three-dimensional Sun vector in the body frame, and a three-axis magnetometer, providing noisy measurements of the magnetic field vector in the body frame. Any two vector sensors could be substituted given that models exist to determine their corresponding measurement values in the inertial frame. This constraint on possible sensors is necessary to meet the requirements for the solution of the classical Wahba's problem. Two extended Kalman filters (EKF) were developed to establish a performance reference based on a "traditional" approach to this nonlinear problem. One filter takes rotational rate measurements from rate sensors, while a second EKF formulation relies on the differencing of quaternions to provide rate "measurements" to the filter. Both were demonstrated to successfully estimate attitude motion of the rotating object, albeit with a performance penalty when using differencing in place of true measurements. Breaking from the traditional approach, two additional algorithms were designed and implemented based on the recently developed unscented Kalman filter (UKF). This was an attempt to capitalize on the superior mean and covariance propagation properties of the UKF relative to the EKF, to achieve better performance given the constraints of this problem. It is the first known application of the UKF to an attitude determination problem. Again, one approach relied on differencing of quaternions to provide rate information, while the second UKF-based algorithm accepted rates from sensors. The UKF-based algorithms were demonstrated to not only successfully estimate attitude motion, but to outperform the EKF algorithms. In one case, the UKF algorithm that incorporates rate measurements produced a 39 percent improvement in the angular error of the rocket spin axis with respect to the EKF, which also incorporated rate measurements All four filters relied on a simplified embedded dynamic model, and the algorithms developed here should be successful in any attitude determination application assuming the appropriate measurements are available, the filter is tuned for the expected motion, and sufficiently small time steps are used. The application of interest for this work was sounding rocket attitude determination, and the filters were tested against simulated rocket rotational motion, at rotational rates about the principle spin axis up to 225 revolutions per minute. For the same baseline example for which results were described above, the UKF filter achieved a performance improvement on the order of 47 percent, when comparing an overall figure of merit, relative to the best EKF performance. This figure of merit is a measure of estimation performance for both attitude and rotational rates. This translates, in terms of attitude determination, into a mean pointing accuracy of the spin vector of 1.97 degrees from true as opposed to 3.24 degrees, an improvement in spin axis determination of approximately 39 percent. In contrast, the spin vector accuracy achieved using only Gauss-Newton error minimization was on the order of 3.28 degrees. Similar significant improvements were achieved for the remaining two axes and the rotational rate estimates as well. The performance of the new UKF-based algorithm was found to be well-behaved at widely varying sensor accuracies, measurement intervals, and in the case where there is an extended loss of measurements. This robust performance establishes this algorithm as a viable alternative to the traditional EKF approach and provides an algorithm suitable for implementation using low cost sensors.

12.2 Recommendations for Further Research

While this effort demonstrated the feasibility of estimating rotational motion of a highly dynamic vehicle, with or without the use of rate measurements, additional work may be envisioned for the development and implementation of this UKF filter algorithm. Despite the simulation of "representative" motion undertaken in this work, true rocket motion displays many idiosyncrasies that cannot be simulated in a straightforward manner. These are the very things that make analytical derivation of a high fidelity model difficult. As such, testing of the algorithm against real attitude data would allow a definitive analysis of how well it does with respect to the uncertainties of real-world motion. The difficulty, of course, is having a "truth" with which to compare in order to judge performance. To successfully accomplish such a validation, it is necessary to obtain data from a rocket flight with a highly accurate (expensive) attitude determination system on board, and measurements compatible with the algorithm developed here. In this way, the high-accuracy solution could be used as a "truth" against which the algorithm solution could be compared. Obviously this data set will not be easy to obtain, but is certainly worth investigating as a next developmental and validation step for a sole source, low-cost, attitude determination system based on the algorithm developed here.

The test cases simulated here all assume a Gaussian noise profile. Additional work is warranted to evaluate how this UKF algorithm performs when the noise on the sensor measurements is not purely additive white Gaussian noise (AWGN), but has a "random walk" or other characteristic. Some insight into this problem, with respect to Kalman filters in general, is given by Grewal et al. who suggest appending a new variable to the state vector and modifying the Φ, Q_k, and H matrices accordingly [87]. A similar approach should be investigated for the UKF. Since the UKF has a parameter, β, that is optimized for a Gaussian distribution when its value is 2 [68], varying this value may make the UKF better suited to other statistical distributions. At the time of this writing, not much research appears to have been done in this area.

As noted early on in this work, the development of low cost sensors is ongoing. One intriguing development is the recent availability of low cost accelerometers. These provide a potential additional source of rate information that could replace, or augment, that provided by the rate gyroscopes assumed herein for the baseline sensor suite. Since these could be easily included in a rocket sensor suite, the intelligent incorporation of their outputs should be investigated.

Further research is also warranted into characterizing the tuning of the model parameters. While an empirical process is demonstrated in this work, where simulation data is available, a formal process for tuning the parameters in the absence of such data would be helpful. Fortunately, the accuracy of real sensor measurements may be estimated pre-launch, thereby mitigating some of the uncertainty in the tuning process. This leaves the overall process noise and the time constants of the rotational motion to be determined. These are functions of the predicted rotational motion, the sensor characteristics, and the lack of fidelity in the embedded dynamic model. If data is to be post-processed, values that minimize the mean of the residual over the duration of the data set might be used. Other techniques should be investigated for use in a real-time implementation.

Finally, the code, as implemented here, has a decidedly developmental aspect to it with many "features" included for user-interaction, data analysis and graphical display of values. While this is particularly well-suited for use as a developmental test bed for evaluating a wide variety of scenarios, for implementation in an actual application, much work can be done to streamline the software to radically decrease execution times. Despite the fact that MATLAB is an exceptional product for engineering development, much faster alternative languages and programming structures are available to the system engineer incorporating this algorithm. Such a streamlining effort would go a long way toward realizing the potential of this algorithm as a real-time process.

13.0 References

13.1 Literature Cited

[1] B. Bull, “A Real Time Differential GPS Tracking System for NASA Sounding Rockets,” in *Proceedings of ION-GPS 2000, Salt Lake City, Utah, 19-22 September 2000.* Institute of Navigation, 2000.

[2] B. Bull, “This is Rocket Science: Multiple Payload Tracking in Space,” GPS World, pp. 22-32, Oct. 2000.

[3] O. L. Cooper, “Rocket Trajectory Analysis from Telemetered Acceleration and Attitude Data,” in International Foundation for Telemetering, 1965.

[4] P. J. Herbert, “Determination of the Attitude of the Skylark Rocket from a Conical Scanning Sun-Slit and Magnetometers,” *Royal Aircraft Establishment Technical Memorandum Space 109, May 1968.*

[5] J. A. Koehler, W. S. C. Brooks and A. Sil, “Sounding Rocket Attitude Determination,” *Canadian Aeronautics and Space Journal,* 22(3): 129-133, May – June 1976.

[6] J. M. Simpson, electronic mail from Joel M. Simpson, Associate Branch Head, GNC SEB at Wallops Flight Facility, Goddard Space Flight Center, to Gary R. Sellhorst regarding research interests at NASA’s Wallops Flight facility, 28 September 2000.

[7] Sounding Rockets Program Office, Suborbital and Special Orbital Projects Directorate, “Sounding Rocket Program Handbook, 810-HB-SRP,” Wallops Island, VA: NASA Goddard Space Flight Center, Wallops Flight Facility, July 2001, http://www.nsroc.com/front/what/SRHB.pdf (30 Oct 2003)

[8] N. Brown, electronic mail from Neal Brown, Director Alaska Space Grant Program, to Joe Hawkins and others regarding data analysis following HEX mission, April 2003.

[9] Student Rocket Project 4 Website, Department of Electrical and Computer Engineering, University of Alaska Fairbanks, Fairbanks, Alaska, http://www.uaf.edu/asgp/ssrp4.htm (16 October 2003).

[10] Department of Astronautics Research, Department of Astronautics at the United States Air Force Academy, Colorado Springs, CO, http://www.usafa.af.mil/dfas/research/index.htm (16 October 2003).

[11] J. L. Marins, “An Extended Kalman Filter for Quaternion-Based Attitude Estimation,” *Engineer Degree’s Thesis*, Naval Postgraduate School, Monterey, CA, September 2000.

[12] J. L. Marins, X. Yun, E. R. Bachmann, R. B. McGhee and M. J. Zyda, “An Extended Kalman Filter for Quaternion-Based Orientation Estimation Using MARG Sensors,” in *Proceedings of the International Conference on Intelligent Robots and Systems, Maui, Hawaii, 29 October – 3 November 2001.* Institute of Electrical and Electronics Engineers, 2001.

[13] R. B. McGhee, E. R. Bachmann, X. P. Yun and M. J. Zyda, “Singularity-Free Estimation of Rigid Body Orientation from Earth Gravity and Magnetic Field Measurements,” Monterey, CA: Naval Postgraduate School, 2002.

[14] D. Gebre-Egziabher, G. H. Elkaim, J. D. Powell and B. W. Parkinson, "A Gyro-Free Quaternion-Based Attitude Determination System Suitable for Implementation Using Low Cost Sensors," in *Proceedings of the 2000 IEEE Position Location and Navigation Symposium, Gaithersburg, Maryland, 12 –16 March 2000.* Institute of Electrical and Electronics Engineers, 2000.

[15] E. R. Bachmann, I. Duman, U.Y. Usta, R. B. McGhee, X. P. Yun and M. J. Zyda, "Orientation Tracking for Humans and Robots Using Inertial Sensors," *Proceedings of 1999 Symposium on Computational Intelligence in Robotics & Automation*, Monterey, CA, November 1999.

[16] P. Zarchan, *Tactical and Strategic Missile Guidance*, Washington, D.C.: American Institute of Aeronautics and Astronautics, Inc., 1990.

[17] P. Zarchan and H. Musoff, *Fundamentals of Kalman Filtering: A Practical Approach.* Reston, VA: American Institute of Aeronautics and Astronautics, Inc., 2000.

[18] J. R. Wertz, "Attitude Geometry," *Spacecraft Attitude Determination and Control*, ed. J. R. Wertz, Dordrecht, Holland: D. Reidel Publishing Company, 1978.

[19] P. M. Smith, "Single-axis Attitude Determination Methods: Methods for Spinning Spacecraft," *Spacecraft Attitude Determination and Control*, ed. J. R. Wertz, Dordrecht, Holland: D. Reidel Publishing Company, 1978.

[20] S. T. Lai, "Attitude Determination of a Spinning and Tumbling Rocket Using Data From Two Orthogonal Magnetometers, AFGL-TR-81-0108," *Air Force Geophysics Laboratory Technical Report 81-0108, 8 April 1981.*

[21] M. Challa and G. Natanson, "Effects of Magnetometer Calibration and Maneuvers on Accuracies of Magnetometer-Only Attitude-and-Rate Determination"" in *Proceedings of the AAS/GSFC 13th International Symposium on Space Flight Dynamics, Greenbelt, Maryland, 11 – 15 May 1998.* National Aeronautics and Space Administration, 1998.

[22] J. L. Crassidis and F. L. Markley, "An MME-Based Attitude Estimator Using Vector Observations," in *Proceedings of the Flight Mechanics and Estimation Theory Symposium,* NASA Conference Publication No. 3299, Goddard Space Flight Center, Greenbelt, May 1995.

[23] J. W. Murrell, "Precision Attitude Determination for Multimission Spacecraft," *AIAA Paper 78-1248.* August 1978.

[24] E. J. Lefferts, F. L. Markley, M. D. Shuster, "Kalman Filtering for Spacecraft Attitude Estimation," *Journal of Guidance*, 5(5): 417-429, September – October 1982.

[25] P. W. Fortescue and J. P. W. Stark, *Spacecraft Systems Engineering*, Chichester, England: John Wiley & Sons Ltd., 1991.

[26] G. M. Siouris, *Aerospace Avionics Systems: A Modern Synthesis*, San Diego, CA: Academic Press, Inc., 1993.

[27] M. H. Kaplan, *Modern Spacecraft Dynamics and Control*, New York, NY: John Wiley & Sons, Inc., 1976.

[28] R. R. Bate, D. D. Mueller and J. E. White, *Fundamentals of Astrodynamics*, New York, NY. Dover Publications, Inc., 1971.

[29] V. I. Feodosiev and G. B. Siniarev, *Introduction to Rocket Technology*, New York, NY: Academic Press, 1959.

[30] W. T. Thomson, *Introduction to Space Dynamics*, New York, NY: Dover Publications, Inc., 1986.

[31] W. E. Wiesel, *Spaceflight Dynamics*, New York, NY: McGraw-Hill Book Company, 1989.

[32] L. Meirovitch, *Methods of Analytical Dynamics*, New York, NY: McGraw-Hill Publishing Company, 1970.

[33] B. N. Agrawal, *Design of Geosynchronous Spacecraft*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1986.

[34] J. A. Holt, "The Application of Kalman Filtering to the Attitude Determination of Spinning Space Vehicles," *Journal of the British Interplanetary Society*, 26(6): 348-363, June 1973.

[35] M. D. Shuster and S. D. Oh, "Three-Axis Attitude Determination form Vector Observations," *AIAA Journal of Guidance and Control*, 4(1): 70-77, January 1981.

[36] M. D. Shuster, "A Survey of Attitude Representations,“ *Journal of the Astronautical Sciences, Vol. 41, No. 4, October-December 1993, pp. 439-517*

[37] F. L. Markley, "Parameterization of the Attitude," *Spacecraft Attitude Determination and Control*, ed. J. R. Wertz, Dordrecht, Holland: D. Reidel Publishing Company, 1978.

[38] J. B. Kuipers, *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace, and Virtual Reality*, Princeton, NJ: Princeton University Press, 1999.

[39] G. A. Smith, "Four Methods of Attitude Determination for Spin-Stabilized Spacecraft With Applications and Comparative Results," *NASA Technical Report R-445*, Washington, D. C.: National Aeronautics and Space Administration, 1975.

[40] V. A. Chobotov, *Spacecraft Attitude Dynamics and Control*, Malabar, FL: Krieger Publishing Company, 1991.

[41] R. W. Hogg, A. L. Rankin, S. I. Roumeliotis, M. C. McHenry, D. M. Helmick, C. F. Bergh and L. Matthies, "Algorithms and Sensors for Small Robot Path Following," in *Proceedings of the 2002 IEEE International Conference on Robotics and Automation, Washington, DC, May 2002.*

[42] G. K. Tandon, "Appendix E: Coordinate Transformations," *Spacecraft Attitude Determination and Control*, ed. J. R. Wertz, Dordrecht, Holland: D. Reidel Publishing Company, 1978.

[43] M. G. Coutinho, *Dynamic Simulations of Multibody Systems*, New York, NY: Springer-Verlag New York, Inc., 2001.

[44] L. Fallon III, "Appendix D: Quaternions," *Spacecraft Attitude Determination and Control*, ed. J. R. Wertz, Dordrecht, Holland: D. Reidel Publishing Company, 1978.

[45] T. R. Kane, P. W. Likins and D. A. Levinson, *Spacecraft Dynamics*, New York, NY: McGraw-Hill Book Company, 1983.

[46] L. Davis, J. W. Follin, and L. Blitzer, *Exterior Ballistics of Rockets*, Princeton, NJ: D. Van Nostrand Company, Inc., 1958.

[47] F. R. Gantmakher and L. M. Levin, *The Flight of Uncontrolled Rockets*, New York, NY: The Macmillan Company, 1964.

[48] E. D. Kaplan, J. L. Leva and M. S. Pavloff, "Fundamentals of Satellite Navigation," *Understanding GPS: Principles and Applications,* ed. E. D. Kaplan, Norwood, MA: Artech House, 1996.

[49] J. S. Eterno, "Attitude Determination and Control," *Space Mission Analysis and Design, 3rd Edition*, ed. J. R. Wertz and W. J. Larson, Torrance, CA: Microcosm Press and The Netherlands: Kluwer Academic Publishers Group, 1999.

[50] E. I. Reeves, "Spacecraft Design and Sizing," *Space Mission Analysis and Design, 3rd Edition*, ed. J. R. Wertz and W. J. Larson, Torrance, CA: Microcosm Press and The Netherlands: Kluwer Academic Publishers Group, 1999.

[51] G. M. Lerner, "Attitude Hardware: Horizon sensors," *Spacecraft Attitude Determination and Control*, ed. J. R. Wertz, Dordrecht, Holland: D. Reidel Publishing Company, 1978.

[52] G. M. Lerner, "Attitude Hardware: Sun sensors," *Spacecraft Attitude Determination and Control,* ed. J. R. Wertz, Dordrecht, Holland: D. Reidel Publishing Company, 1978.

[53] F. Tohyama and N. Yamamoto, *The Final Report for SRP-4 Rocket Experiment: Measurement of Geomagnetic Field and Attitude of Rocket by Fluxgate Magnetometer and Sun Sensor*, Tokai University, Shizuoka, Japan, 2003.

[54] L. Fallon III, "Attitude Hardware: Gyroscopes," *Spacecraft Attitude Determination and Control,* ed. J. R. Wertz, Dordrecht, Holland: D. Reidel Publishing Company, 1978.

[55] B. T. Blaylock, "Attitude Hardware: Magnetometers," *Spacecraft Attitude Determination and Control*, ed. J. R. Wertz, Dordrecht, Holland: D. Reidel Publishing Company, 1978.

[56] H. Yamaguchi, electronic mail from Hiroyuki Yamaguchi, Tokai Student Rocket Project, to Mark Charlton detailing accuracy of the SRP-4 magnetometer, 16 October 2003.

[57] J. Deutschmann and I. Bar-Itzhack, "Attitude and Trajectory Estimation Using Earth Magnetic Field Data," in *Proceedings of the Flight Mechanics and Estimation Theory Symposium,* NASA Conference Publication No. 3333, Goddard Space Flight Center, Greenbelt, May 1996.

[58] A. N. Steinberg and C. L. Bowman, "Revisions to the JDL Data Fusion Model," *Handbook of Multisensor Data Fusion*, ed. D. L. Hall and J. Llinas, Boca Raton, FL: CRC Press LLC, 2001.

[59] J. K. Uhlmann, "Introduction to the Algorithmics of Data Association in Multiple-Target Tracking," *Handbook of Multisensor Data Fusion*, ed. D. L. Hall and J. Llinas, Boca Raton, FL: CRC Press LLC, 2001.

[60] D. L. Hall and J. Llinas, "Multisensor Data Fusion," *Handbook of Multisensor Data Fusion*, ed. D. L. Hall and J. Llinas, Boca Raton, FL: CRC Press LLC, 2001.

[61] A. Perrella, C. Glover, and S. Waugh, "Does Extra Information Always Help in Data Fusion?," Proceedings of SPIE, *Signal and Data Processing of Small Targets 1999*, pp. 344-354, 20-22 July 1999.

[62] J. Llinas, "Assessing the Performance of Multisensor Fusion Processes" *Handbook of Multisensor Data Fusion*, ed. D. L. Hall and J. Llinas, Boca Raton, FL: CRC Press LLC, 2001.

[63] D. L. Hall and A. N. Steinberg, "Dirty Secrets in Multisensor Data Fusion," *Handbook of Multisensor Data Fusion*, ed. D. L. Hall and J. Llinas, Boca Raton, FL: CRC Press LLC, 2001.

[64] M. T. Hagan, H. B. Demuth and M. Beale, *Neural Network Design*, Boston, MA: PWS Publishing Company, 1996.

[65] B. Widrow and S. Stearns, *Adaptive Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1985.

[66] G. Wahba, "Problem 65-1, A least Squares Estimate of Satellite Attitude," *SIAM Review*, 8(3): 384-386. June 1966.

[67] S. Haykin, *Kalman Filtering and Neural Networks*, New York, NY: John Wiley & Sons, Inc., 2001.

[68] E. A. Wan and R. van der Merwe, "The Unscented Kalman Filter," *Kalman Filtering and Neural Networks*, ed. S. Haykin, New York, NY: John Wiley & Sons, Inc., 2001.

[69] S. Julier and J. K. Uhlmann, "Data Fusion in Nonlinear Systems," *Handbook of Multisensor Data Fusion*, ed. D. L. Hall and J. Llinas, Boca Raton, FL: CRC Press LLC, 2001.

[70] E. A. Wan and A. T. Nelson, "Dual Extended Kalman Filter Methods," *Kalman Filtering and Neural Networks*, ed. S. Haykin, New York, NY: John Wiley & Sons, Inc., 2001

[71] M. S. Grewal and A. P. Andrews, *Kalman Filtering: Theory and Practice Using MATLAB, second edition*, New York, NY: John Wiley & Sons, Inc., 2001.

[72] R. G. Brown and P. Y. C. Hwang, *Introduction to Random Signals and Applied Kalman Filtering with MATLAB Exercises and Solutions, Third Edition*, New York, NY: John Wiley & Sons, INC., 1997.

[73] R. E. Kalman, "A New Approach to Linear Filtering and Prediction Problems," *ASME Journal of Basic Engineering*, Vol. 82, pp. 34-45, 1960.

[74] J. L. Leva, M. U. de Haag and K. V. Van Dyke, "Performance of Standalone GPS," *Understanding GPS: Principles and Applications*, ed. E. D. Kaplan, Norwood, MA: Artech House, 1996.

[75] A. Gelb, *Applied Optimal Estimation*, Cambridge, MA: The M. I. T. Press, 1974.

[76] M. Foss and G. J. Geier, "Integration of GPS With Other Sensors," *Understanding GPS: Principles and Applications*, ed. E. D. Kaplan, Norwood, MA: Artech House, 1996.

[77] P. Axelrad and R. G. Brown, "GPS Navigation Algorithms," *Global Positioning System: Theory and Applications, Vol. I*, ed. B. W. Parkinson and J. J. Spilker, Washington D. C.: American Institute of Aeronautics and Astronautics, Inc., 1996.

[78] G. Chen, "Preface," *Approximate Kalman Filtering*, ed. G. Chen, Singapore: World Scientific Publishing Co. Pte. Ltd., 1993.

[79] K. Chui and G. Chen, *Kalman Filtering with Real-Time Applications*, New York, NY: Springer-Verlag, 1999.

[80] R. G. Brown, *Introduction to Random Signal Analysis and Kalman Filtering*, New York, NY: John Wiley & Sons, Inc., 1983.

[81] P. Hanlon and P. Maybeck, "Multiple-Model Adaptive Estimation Using a Residual Correlation Kalman Filter Bank," *IEEE Transactions on Aerospace and Electronic systems*, Vol. 36, No. 2, April 2000, pp 393-406.

[82] K. V. Ramachandra, *Kalman Filtering Techniques for Radar Tracking*, New York, NY: Marcel Dekker, Inc., 2000.

[83] J. F. Brule', "Fuzzy Systems – A tutorial," http://www.austinlingks.com/Fuzzy/tutorial.html, 1985.

[84] R. van der Merwe and E. A. Wan, "Efficient Derivative-Free Kalman Filters for Online Learning," in *Proceedings of the 2001 European Symposium on Artificial Neural Networks*, Bruges, Belgium, 25-27 April 2001, D-Facto Public., 2001.

[85] T. P. Yunck, "Orbit Determination," *Global Positioning System: Theory and Applications, Vol. II*, ed. B. W. Parkinson and J. J. Spilker, Washington D. C.: American Institute of Aeronautics and Astronautics, Inc., 1996.

[86] T. Logsdon, *The Navstar Global Positioning System.* New York, NY: Van Nostrand Reinhold, 1992.

[87] M. S. Grewal, L. R. Weill, and A. P. Andrews, *Global Positioning Systems, Inertial Navigation, and Integration*, New York, NY: John Wiley & Sons, Inc., 2001.

13.2 Other References

[88] S. F. Andrews and W. M. Morgenstern, "Initial Flight Results of the TRMM Kalman Filter," in *Proceedings of the AAS/GSFC 13th International Symposium on Space Flight Dynamics, Greenbelt, Maryland, 11 – 15 May 1998.* National Aeronautics and Space Administration, 1998.

[89] R. Antony, "Data Fusion Automation: A Top-Down Perspective," *Handbook of Multisensor Data Fusion*, ed. D. L. Hall and J. Llinas, Boca Raton, FL: CRC Press LLC, 2001.

[90] R. M. Beard and M. Plett, "Attitude Dynamics: Spacecraft Nutation," *Spacecraft Attitude Determination and Control*, ed. J. R. Wertz, Dordrecht, Holland: D. Reidel Publishing Company, 1978.

[91] R. R. Brooks and L. Grewe, "Data Registration," *Handbook of Multisensor Data Fusion*, ed. D. L. Hall and J. Llinas, Boca Raton, FL: CRC Press LLC, 2001.

[92] D. Catlin, "Fisher Initialization in the Presence of Ill-Conditioned Measurements," *Approximate Kalman Filtering*, ed. G. Chen, Singapore: World Scientific Publishing Co. Pte. Ltd., 1993.

[93] M. Challa, G. Natanson, J. Deutschmann and K. Galal, "A PC-Based Magnetometer-Only Attitude and Rate Determination System for Gyroless Spacecraft," in *Proceedings of the Flight Mechanics and Estimation Theory Symposium,* NASA Conference Publication No. 3299, Goddard Space Flight Center, Greenbelt, May 1995.

[94] M. Challa and C. Wheeler, "Accuracy Studies of a Magnetometer-Only Attitude-and-Rate-Determination System," in *Proceedings of the Flight Mechanics and Estimation Theory*

Symposium, NASA Conference Publication No. 3333, Goddard Space Flight Center, Greenbelt, May 1996.

[95] L. C. Chen and J. R. Wertz, "Geometrical Basis for Attitude Determination," *Spacecraft Attitude Determination and Control*, ed. J. R. Wertz, Dordrecht, Holland: D. Reidel Publishing Company, 1978.

[96] L. Fallon III, "State Estimation Attitude Determination Methods: Introduction to Estimation Theory," *Spacecraft Attitude Determination and Control*, ed. J. R. Wertz, Dordrecht, Holland: D. Reidel Publishing Company, 1978.

[97] M. Forrest, T. Spracklen and N. Ryan, "An Inertial Navigation Data Fusion System Employing an Artificial Neural Network as the Data Integrator," *Proceedings of the Institute of Navigation 2000 National Technical Meeting: 2000 – Navigating into the New Millennium*, January 26-28, 2000, Anaheim, CA.

[98] Q. Gan and C. J. Harris, "Comparison of Two Measurement Fusion Methods for Kalman-Filter-Based Multisensor Data Fusion," *IEEE Transactions on Aerospace and Electronic systems*, Vol. 37, No. 1, January 2001, pp 273-280.

[99] R. L. Greenspan, "GPS and Inertial Integration," *Global Positioning System: Theory and Applications, Vol. II*, ed. B. W. Parkinson and J. J. Spilker, Washington D. C.: American Institute of Aeronautics and Astronautics, Inc., 1996.

[100] S. G. Hotovy, "State Estimation Attitude Determination Methods: Observation Models," *Spacecraft Attitude Determination and Control*, ed. J. R. Wertz, Dordrecht, Holland: D. Reidel Publishing Company, 1978.

[101] S. G. Hotovy, "State Estimation Attitude Determination Methods: State Vectors," *Spacecraft Attitude Determination and Control*, ed. J. R. Wertz, Dordrecht, Holland: D. Reidel Publishing Company, 1978.

[102] S. Julier and J. K. Uhlmann, "General Decentralized Data Fusion with Covariance Intersection (CI)," *Handbook of Multisensor Data Fusion*, ed. D. L. Hall and J. Llinas, Boca Raton, FL: CRC Press LLC, 2001.

[103] G. M. Lerner, "Data Validation and Smoothing," *Spacecraft Attitude Determination and Control*, ed. J. R. Wertz, Dordrecht, Holland: D. Reidel Publishing Company, 1978.

[104] G. M. Lerner, "Three-Axis Attitude Determination," *Spacecraft Attitude Determination and Control*, ed. J. R. Wertz, Dordrecht, Holland: D. Reidel Publishing Company, 1978.

[105] F. L. Markley, "Attitude Dynamics: Equations of Motion," *Spacecraft Attitude Determination and Control*, ed. J. R. Wertz, Dordrecht, Holland: D. Reidel Publishing Company, 197.

[106] F. L. Markley, "Response to Torques," *Spacecraft Attitude Determination and Control*, ed. J. R. Wertz, Dordrecht, Holland: D. Reidel Publishing Company, 1978.

[107] T. Papis and C. W. Bullard, "Determination of the Attitude of a Spinning Rocket Under Thrust with Statistically Varied Inputs, AIAA Paper No. 67-536," in *Proceedings of the AIAA Guidance, Control and Flight Dynamics Conference, Huntsville, Alabama, August 14 – 16, 1967.*

[108] M. Plett, "The Earth's Magnetic Field," *Spacecraft Attitude Determination and Control*, ed. J. R. Wertz, Dordrecht, Holland: D. Reidel Publishing Company, 1978.

[109] R. Rogers, *Applied Mathematics in Integrated Navigation Systems*. Reston, VA: American Institute of Aeronautics and Astronautics, Inc., 2000.

[110] J. N. Rowe, "Modeling the Positions of the Sun, Moon, and Planets," *Spacecraft Attitude Determination and Control*, ed. J. R. Wertz, Dordrecht, Holland: D. Reidel Publishing Company, 1978.

[111] S. Roweis and Z. Ghahramani, "Learning Nonlinear Dynamical Systems Using the Expectation-Maximization Algorithm," *Kalman Filtering and Neural Networks*, ed. S. Haykin, New York, NY: John Wiley & Sons, Inc., 2001.

[112] P. M. Smith, "Single-axis Attitude Determination Methods," *Spacecraft Attitude Determination and Control*, ed. J. R. Wertz, Dordrecht, Holland: D. Reidel Publishing Company, 1978.

[113] C. B. Spence, Jr. and F. L. Markley, "Attitude Prediction: Attitude Propagation," *Spacecraft Attitude Determination and Control*, ed. J. R. Wertz, Dordrecht, Holland: D. Reidel Publishing Company, 1978

[114] P. Vanicek and M. Omerbasic, "Does a Navigation Algorithm Have To Use a Kalman Filter?" *Canadian Aeronautics and Space Journal*, Vol. 45, No. 3, September 1999, pp 292-296.

[115] E. Waltz and D. L. Hall, "Requirements for Derivation of Data Fusion Systems," *Handbook of Multisensor Data Fusion*, ed. D. L. Hall and J. Llinas, Boca Raton, FL: CRC Press LLC, 2001.

[116] J. R. Wertz, "Evaluation and Use of State Estimators," *Spacecraft Attitude Determination and Control*, ed. J. R. Wertz, Dordrecht, Holland: D. Reidel Publishing Company, 1978.

[117] J. R. Wertz, "Introduction," *Spacecraft Attitude Determination and Control*, ed. J. R. Wertz, Dordrecht, Holland: D. Reidel Publishing Company, 1978.

[118] J. R. Wertz, "Introduction to Attitude Dynamics and Control," *Spacecraft Attitude Determination and Control*, ed. J. R. Wertz, Dordrecht, Holland: D. Reidel Publishing Company, 1978.

[119] J. R. Wertz, "State Estimation Attitude Determination Methods: Deterministic Versus State Estimation Attitude Methods," *Spacecraft Attitude Determination and Control*, ed. J. R. Wertz, Dordrecht, Holland: D. Reidel Publishing Company, 1978.

Appendix A
Software Code to Support Research

This appendix contains the software written to implement this research effort. All software was developed using the MATLAB software package. It is written with analysis and design of the filter algorithms in mind and contains many "extra" features that are not necessary when implementing any one of the algorithms for an actual application. One example of this is the extensive use of plots for diagnostic purposes, and to illustrate operation of the various simulations and filters. It is also worthy of note that MATLAB is an exceptional program for design and research, and has many built-in functions that are both extensive and user-friendly. It is not, however, an extremely fast programming method. Only a minimal effort was made to streamline the programming, although choices were made, such as numerical integration approach, which translate into time savings regardless of implementation language. The programming style is certainly not elegant, and many optimizations can be made to speed up execution, to include programming the filter of choice in a more efficient computer language. For all of these reasons, the software as currently written serves its purpose, but is slower than can be expected in an actual application. The following sections describe the various routines used for this effort.

A.1 Attitude_filter.m

Attitude_filter.m is the main MATLAB script for this software implementation. It calls the other software scripts and functions defined in this appendix at the appropriate time to simulate the "true" motion and the noisy measurements of the motion. Next it processes the noisy measurements through the Gauss-Newton error minimization routine and through the filtering algorithm selected by the user. This is the "master" script that is run by the user.

```
%Attitude_filter.m -- main script to call m-files to execute filtering of noisy attitude sensor data.
%Implements both a traditional Extended Kalman Filter and an "Unscented" Kalman Filter.
%Inclusion of rate measurements is a user-selectable option.

%Written by Mark Charlton. Last updated 21 October 03.

format compact
clc;

'This program calls the following m-files to simulate the Kalman filtering of noisy attitude sensor'
'data for a sounding rocket subject to rotational motion. Attitude_sim.m simulates the rotational'
'motion as well as the noisy measurements of that motion. A traditional Extended Kalman Filter (EKF)'
'is implemented in norates_ekf.m and in rates_ekf.m while an "Unscented" Kalman Filter (UKF) is'
```

```
'implemented in norates_ukf.m and in rates_ukf.m.  Attitude_sim.m generates the simulated true three-'
'dimensional motion of the object, as well as simulated attitude sensor measurements in the form of'
'3D vectors.  It also generates simulated noisy rate measurements.  These measurements have user-'
'defined sigmas for the measurement noise.  Norates_ekf.m and norates_ukf.m filter the data generated'
'by attitude_sim.m as if no rate measurements were available and plot filter results vs. truth data.'
'These m-files also compute the Mean-Square-Error (MSE) for each technique to aid in judging algorithm'
'performance. Rates_ekf.m and rates_ukf.m do similar filtering and performance measurement, only now'
'they include the simulated noisy rate measurements. From attitude_filter.m, the user may choose to'
'generate a new set of attitude_sim.m generated data for each filter run, or may choose to continue'
'filtering the same data while changing parameters of the filter.  This aids in investigating both how'
'a given set of filter parameters performs on a set of data as well as how, for a given set of data,'
'the filter performance varies with changes in parameters. Reasonable Default values are available'
%for each parameter.'
' '
'press "return" to continue...'
pause;clc;close;

process=1;
generate=1;

while process==1
  clc;
  'You may either call attitude_sim.m to generate a new set of simulated data by'
  'entering "1", or process the existing data again by entering "0".'
  ' '
  'If this is the first run, you must generate data (enter "1").'
  ' '
  generate=input('Enter "1" or "0" ("return" defaults to same data):  ');
  if isempty(generate)
    generate=0
  end
  if generate==1
    clear all
    attitude_sim                        %call m-file to generate truth and noisy measurements
    generate=1;

    clc

    %****** once data is generated, calculate attitude quaternions from noisy measurements ******
    if generate==1              %indicates new data has been generated, and minimization should be accomplished

      '****** Gauss-Newton error minimization ******'
      'Press "return" to calculate the attitude quaternion from the noisy'
      'attitude measurements using a Gauss-Newton error minimization...'
```

```
pause;close;clc;

vref=[ ];
vrot=[ ];
q_min=[ ];
q_min_dot=[ ];
qmin1init=0;      %initial estimate of attitude quaternion
qmin2init=0;
qmin3init=0;
qmin4init=1;

for i=1:count;
  vref(1:6,i)=[SUNVEC(1:3,i);MAGVEC(1:3,i)];                    %form reference vector from known inertial vectors

  unitsunmeas(1:3,i)=sunmeas(1:3,i)/norm(sunmeas(1:3,i));       %insure noisy sun measurement is unit vector
  unitmagmeas(1:3,i)=magmeas(1:3,i)/norm(magmeas(1:3,i));       %insure noisy mag measurement is unit vector

  vrot(1:6,i)=[unitsunmeas(1:3,i);unitmagmeas(1:3,i)];          %form vector to be rotated by "best" quaternion

  if i<2
    q_init=[qmin1init qmin2init qmin3init qmin4init]';
  else
    q_init=q_min(1:4,i-1);
  end

  if norm(sunmeas(1:3,i))<.01               %default values in case of loss of data (no sun vector measurement)
    q_min(1:4,i)=[qmin1init qmin2init qmin3init qmin4init]';
    A(1:4,1:6,i)=A(1:4,1:6,i-1);
    check(i)=check(i-1);
    err(i)=err(i-1);
  elseif norm(magmeas(1:3,i))<.01           %default values in case of loss of data (no mag field measurement)
    q_min(1:4,i)=[qmin1init qmin2init qmin3init qmin4init]';
    A(1:4,1:6,i)=A(1:4,1:6,i-1);
    check(i)=check(i-1);
    err(i)=err(i-1);
  else
    %if measurements exist...proceed with minimization...

    steps=10;                              %defines max allowable iterations for convergence within Gauss_newton
    tol=.01;                               %defines convergence tolerance for Gauss_Newton

    [q_min(1:4,i), A(1:4,1:6,i), check(i), err(i)]=Gauss_newton(vref(1:6,i), vrot(1:6,i), q_init, steps, tol);

    %****** generate noisy measurements of quaternion rate by differencing subsequent quaternions ******
```

```
          %****** these results are used as "derived" rate measurements when filtering is done without  ******
          %****** noisy rate measurements from sensors ******

          if i>1                          %quaternion differencing to get qdot and then omega requires one time step delay!!!
            q_min_dot(1:4,i-1)=(q_min(1:4,i)-q_min(1:4,i-1))/TS;

            %****** calculate body rates corresponding to derived quaternion and quaternion rates ******

            omega1_q_min(i-1)=2*(q_min(4,i-1)*q_min_dot(1,i-1)+q_min(3,i-1)*q_min_dot(2,i-1)-q_min(2,i-1)*q_min_dot(3,i-
1)-q_min(1,i-1)*q_min_dot(4,i-1));
            omega2_q_min(i-1)=2*(-q_min(3,i-1)*q_min_dot(1,i-1)+q_min(4,i-1)*q_min_dot(2,i-1)+q_min(1,i-
1)*q_min_dot(3,i-1)-q_min(2,i-1)*q_min_dot(4,i-1));
            omega3_q_min(i-1)=2*(q_min(2,i-1)*q_min_dot(1,i-1)-q_min(1,i-1)*q_min_dot(2,i-1)+q_min(4,i-1)*q_min_dot(3,i-
1)-q_min(3,i-1)*q_min_dot(4,i-1));
          end
        end
      end

  %****** Calculate standard deviation of error in measured omegas to use in filter R matrices -- not available in real world ******

      stdomega1=std(omega1(1:length(omega1_q_min))-omega1_q_min);
      stdomega2=std(omega2(1:length(omega2_q_min))-omega2_q_min);      %use these when no rate measurements are available
      stdomega3=std(omega3(1:length(omega3_q_min))-omega3_q_min);

      stdomega1meas=std(omega1measerror);
      stdomega2meas=std(omega2measerror);                              %use these when rate measurements are available
      stdomega3meas=std(omega3measerror);

      %****** Plots of derived quaternion components versus true quaternions from omega_sim ******

      plot(T(1:count),q_min(1,1:count),'g+-', T(1:count), Qvec(1,1:count)),grid on;
      title('derived  q1 vs. Euler-based q1');xlabel('time (sec)');ylabel('q1');
      legend('derived q1', ' Euler-based q1');
      pause;close;clc;

      plot(T(1:count),q_min(2,1:count),'g+-', T(1:count), Qvec(2,1:count)),grid on;
      title('derived  q2 vs. Euler-based q2');xlabel('time (sec)');ylabel('q2');
      legend('derived q2', 'Euler-based q2');
      pause;close;clc;

      plot(T(1:count),q_min(3,1:count),'g+-', T(1:count), Qvec(3,1:count)),grid on;
      title('derived  q3 vs. Euler-based q3');xlabel('time (sec)');ylabel('q3');
      legend('derived q3', 'Euler-based q3');
      pause;close;clc;
```

```
plot(T(1:count),q_min(4,1:count),'g+-', T(1:count), Qvec(4,1:count)),grid on;
title('derived  q4 vs. Euler-based q4');xlabel('time (sec)');ylabel('q4');
legend('derived q4', 'Euler-based q4');
pause;close;clc;

%****** rotate each vector using actual and derived attitude quaternion ******

%****** check mean-square-error between vectors ******

'****** Evaluate Mean-Square-Error of rotated vectors using Gauss-Newton derived quaternion ******'
'Press "return" to rotate the reference vector at each time step to the'
'representation in the rotated frame using both the actual rotation quaternion'
'and the quaternion derived from the noisy measurements...once rotation is complete,'
'calculate the mean-square-error between the results.'
pause;close;clc;

GN_mse_error=[ ];

for j=1:count;
   Q_refvec(1:6,j)=qframerot6(Qvec(1:4,j),vref(1:6,j));            %rotate ref vector using true quaternion
   q_minvec(1:6,j)=qframerot6(q_min(1:4,j),vref(1:6,j));           %rotate ref vector using min error quaternion
   GN_mse_error(j)=mean_square_error(q_minvec(1:6,j),Q_refvec(1:6,j));
                                                                   %calculate mse between rotated vectors
end

'The average mse error for the reference vectors rotated using the GN-derived quaternion is: '
mean_GN_mse_error=mean(GN_mse_error)

%****** Plot vectors rotated using derived and actual quaternions *****

plot3(Q_refvec(1,1:count),Q_refvec(2,1:count),Q_refvec(3,1:count),'b');grid on;hold on;
plot3(q_minvec(1,1:count),q_minvec(2,1:count),q_minvec(3,1:count),'m:');
plot3(Q_refvec(4,1:count),Q_refvec(5,1:count),Q_refvec(6,1:count),'r');grid on;hold on;
plot3(q_minvec(4,1:count),q_minvec(5,1:count),q_minvec(6,1:count),'g:');
xlabel('X component');ylabel('Y component');zlabel('Z component');
legend('sun vector, actual q','sun vector, derived q','mag field vector, actual q', 'mag field vector, derived q');
title('3-D plot of reference vectors rotated using actual quaternions and derived quaternions');
pause;close;clc;

%****** Plot mean-square-error between reference vector rotated using actual quaternion ******
%****** and reference vector rotated using derived quaternion.                  ******

plot(T(1:count),GN_mse_error(1:count)),grid on;
```

```
      title('MSE between reference vectors rotated using derived and actual quaternions');
      xlabel('time (sec)');ylabel('MSE');legend('MSE')
      'press "return" to continue...'
      pause;close;clc;
   end
end

'Indicate whether to use rate measurments in this filtering run... '
''
ratemeas=input('Enter 1 if rate measurements are available, 0 if rates are derived from GN:  ')
if isempty(ratemeas)
   ratemeas=0
end

if ratemeas==0

   '****** No rate sensor data is being used... ******'

   '****** Extended Kalman Filter ******'
   ''
   'Press "return" to filter data from attitude_sim.m using a traditional Extended Kalman Filter...'
   ''
   'Press return to advance through various options...'
   pause;clc;close;

   norates_ekf                                  %call m-file to filter data with traditional extended Kalman filter

   '****** Unscented Kalman Filter ******'
   ''
   'press "return" to filter data from attitude_sim.m using an "Unscented" Kalman Filter...'
   pause;clc;close;

   norates_ukf                                  %call m-file to filter data with "unscented" Kalman filter

   'press "return" to continue...'
   pause;clc;close;

elseif ratemeas==1

   '****** Rate sensor data is being used... ******'
   ''
   '****** Extended Kalman Filter ******'
   ''
   'Press "return" to filter data from attitude_sim.m using a traditional Extended Kalman Filter...'
```

```
  ''
  'Press return to advance through various options...'
  pause;clc;close;

  rates_ekf                                   %call m-file to filter data with traditional extended Kalman filter

  '****** Unscented Kalman Filter ******'
  ''
  'press "return" to filter data from attitude_sim.m using an "Unscented" Kalman Filter...'
  pause;clc;close;

  rates_ukf                                   %call m-file to filter data with "unscented" Kalman filter

  'press "return" to continue...'
  pause;clc;close;

end

'*****  Recap of relative performance  *****'
''
'Indicator of overall attitude and rotation rate performance by GN...'
''
overall_GN_perf
''
'Indicator of overall attitude and rotation rate performance by EKF...'
''
overall_ekf_perf
''
'Indicator of overall attitude and rotation rate performance by UKF...'
''
overall_ukf_perf
''
'press "return to continue..."'
pause;close;clc;

'*******************************************************************************'
'*********** End of this run---Do you wish to run another case? *************'
'*******************************************************************************'
''
''
process=input('Enter "1" for yes or "0" for no. Default is to stop.  ');

if isempty (process)
  process=0
  end
end

'end of program'
```

A.2 Attitude_sim.m

Attitude_sim.m is called by attitude_filter.m to simulate true motion of the vehicle and the noisy sensor measurements of that motion. Attitude_sim.m produces "true" Euler angles and "true" attitude quaternions at each time step. The user has the option to enter the desired sensor measurement standard deviations, the initial conditions, the simulation duration, and the type of rate profile desired, among other things.

```
%Attitude_sim- simulates "true" object attitude motion by propagating
%Euler angles forward through the numerical integration of the Euler equations
%using a second-order Runge-Kutta integration. The Euler angular rate profile
%is calculated from a user defined profile of the angular rates in the body-
%fixed frame. The propagated Euler angles are used at each time step to form a
%direction cosine matrix which is used to rotate fixed inertial reference
%vectors into the current body frame. Noisy attitude sensor measurements and noisy
%measurements of angular rate are then simulated by adding random noise with a defined
%sigma corresponding to each sensor. The corresponding attitude quaternion is computed
%at each time step from the propagated Euler angles. The user may select a profile with
%fixed angular rates, linearly ramped angular rates, or rate profiles that asymptotically
%approach a final value.

%Written by Mark Charlton, last modified: 28 October 2003
%References: [40], [30], [11]

clear all;
clc;
format compact
format short

'************ SIMULATION OF "TRUE" OBJECT ATTITUDE MOTION AND NOISY SENSOR MEASUREMENTS
************'
''
'NOTE: All attitude vector components are referenced to an Earth-Centered Inertial reference'
'frame. Therefore positive X is pointed to the first point of Aries, positive Z is out the'
'Earth north pole, and Y completes the right-handed orthogonal triad. When asked to enter'
'values, hit "return" for default value.'
''
```

```
'NOTE: This uses a 3-1-3 Euler angle sequence:  first through psi about the original "z" axis,'
'then through theta about the new "x" axis, finally through phi about the new "z" axis.'
' '

%************ input measurement errors **************
'DATA ENTRY FOR SIMULATION'
' '
%enter angular standard deviation of sun sensor measurement in degrees (assumes equal
%measurement error in all axes)
SIGSUN=input('Please enter the angular standard deviation of the sun sensor measurement in degrees (default = 1.333):  ');
if isempty(SIGSUN)
        SIGSUN=1.333
end

%convert sun sensor sigma to radians
SIGSUN=SIGSUN*pi/180;

%convert sun sensor sigma from angular to absolute measure for unit sun
%vector

SIGSUN=sin(SIGSUN);          %based on unit vector and small angle approximation

%***** for now...SIGMA is same in each axis *****

SIGSUNX=SIGSUN;
SIGSUNY=SIGSUN;
SIGSUNZ=SIGSUN;

%enter angular standard deviation of magnetometer measurement in degrees (assumes equal
%measurement error in all axes)
SIGMAG=input('Please enter the angular standard deviation of the magnetometer measurement in degrees (default = 3.333):  ');
if isempty(SIGMAG)
        SIGMAG=3.333          %based on unit vector and small angle approximation
end

%convert magnetometer sensor sigma to radians
SIGMAG=SIGMAG*pi/180;

%convert magnetometer sensor sigma from angular to absolute measure
%for unit sun vector

SIGMAG=sin(SIGMAG);

%***** for now...SIGMA is same in each axis *****
```

```
SIGMAGX=SIGMAG;
SIGMAGY=SIGMAG;
SIGMAGZ=SIGMAG;

%enter standard deviation of omega sensor measurement in rpm (assumes equal
%measurement error in all axes)
SIGOMEGA=input('Please enter the standard deviation of the rotational rate sensor measurement in rpm (default = 0.333): ');
if isempty(SIGOMEGA)
        SIGOMEGA=0.333
end

%convert omega sensor sigma to radians/sec
SIGOMEGA=SIGOMEGA*2*pi/60;


SIGOMEGA1=SIGOMEGA;
SIGOMEGA2=SIGOMEGA;
SIGOMEGA3=SIGOMEGA;

%************ input reference vectors **************
%Initial reference sun vector X component (from ephemeris given time and
%location)

SUNX=input('Please enter the initial true X component of the sun vector (default = 1)): ');
if isempty(SUNX)
        SUNX=1
end

%Initial reference sun vector Y component (from ephemeris given time and
%location)

SUNY=input('Please enter the initial true Y component of the sun vector (default = 1): ');
if isempty(SUNY)
        SUNY=1
end

%Initial reference sun vector Z component (from ephemeris given time and
%location)

SUNZ=input('Please enter the initial true Z component of the sun vector (default = 1): ');
if isempty(SUNZ)
  SUNZ=1
end
```

```
%Initial reference mag field vector X component (from mag field model given location)

MAGX=input('Please enter the initial true X component of the unit mag field vector (default = -1):  ');
if isempty(MAGX)
  MAGX=-1
end

%Initial reference mag field vector Y component (from mag field model given location)

MAGY=input('Please enter the initial true Y component of the unit mag field vector (default = 1):  ');
if isempty(MAGY)
  MAGY=1
end

%Initial reference mag field vector Z component (from mag field model given location)

MAGZ=input('Please enter the initial true Z component of the unit mag field vector (default = -1):  ');
if isempty(MAGZ)
  MAGZ=-1
end

omega1=[ ];
omega2=[ ];
omega3=[ ];
psi=[ ];
theta=[ ];                    %initialization of matrices
phi=[ ];
psidot=[ ];
thetadot=[ ];
phidot=[ ];
T=[ ];

%************ input initial Euler angles **************

%Input initial Euler angle, psi

psi0=input('Please enter the initial value of the Euler angle "psi" in degrees (default = 15):  ');
if isempty(psi0)
  psi0=15
end
%convert psi0 to radians
psi0=psi0*pi/180;
```

```
psi(1)=psi0;

%Input initial Euler angle, theta

theta0=input('Please enter the initial value of the Euler angle "theta" in degrees (default = 30):  ');
if isempty(theta0)
  theta0=30
end

%convert theta0 to radians
theta0=theta0*pi/180;

theta(1)=theta0;
%Input initial Euler angle, phi

phi0=input('Please enter the initial value of the Euler angle "phi" in degrees (default = 45):  ');
if isempty(phi0)
  phi0=45
end

%convert phi0 to radians
phi0=phi0*pi/180;

phi(1)=phi0;

%************ input total simulation time **************

total_time=input('Please enter the desired length of simulation in seconds (default = 40):  ');
if isempty(total_time)
  total_time=40
end

%****** determine desired omega profile and call appropriate omega generator *******

'The following generation methods are available: 1 = fixed body rates, 2 = linearly ramped body rates,'
'3 = exponentially derived body rates, 4 = step function / pulse'
''
method=input('Please enter the number of the desired generation method:  ')

if isempty(method)
            method=3
end

if method == 1
```

```
   %****** call omegagen_fix to generate arrays of angular rates about the rotated frame axes ******

   [omega1, omega2, omega3,TS]=omegagen_fix(total_time);
end

if method == 2

   %****** call omegagen_ramp to generate arrays of angular rates about the rotated frame axes ******

   [omega1, omega2, omega3,TS]=omegagen_ramp(total_time);
end

if method == 3

   %****** call omegagen_exp to generate arrays of angular rates about the rotated frame axes ******

   [omega1, omega2, omega3,TS]=omegagen_exp(total_time);
end

if method == 4

   %****** call omegagen_step to generate arrays of angular rates about the rotated frame axes ******

   [omega1, omega2, omega3,TS]=omegagen_step(total_time);
end

%****** Numerical integration of Euler equations ******
%****** Propagates Euler angles forward using body rates and initial cond ******

count=1;                        %initializes counter
H=TS/10;                          %set integration time step
T=[ ];
T(1)=0;                        %array of times of each measurement

%****** Calculate initial Euler angle derivatives ******

if abs(theta)<eps
   theta=1e-4;                    %prevent divide by zero / singularity
end
if abs(theta(count)-pi)<eps
   if sign(theta(count)-pi)==1
      theta(count)=theta(count)+1e-4;
   else
```

```
    theta(count)=theta(count)-1e-4;
  end
end

psidot(count)=1/sin(theta(count))*(sin(phi(count))*omega1(count)+cos(phi(count))*omega2(count));
thetadot(count)=1/sin(theta(count))*(cos(phi(count))*sin(theta(count))*omega1(count)-
sin(phi(count))*sin(theta(count))*omega2(count));
phidot(count)=1/sin(theta(count))*(-sin(phi(count))*cos(theta(count))*omega1(count)-
cos(phi(count))*cos(theta(count))*omega2(count)+sin(theta(count))*omega3(count));

SUNVEC=[ ];
MAGVEC=[ ];
sunvec=[ ];
magvec=[ ];
q1=[ ];
q2=[ ];
q3=[ ];
q4=[ ];
Qvec=[ ];
sunmeas=[ ];
magmeas=[ ];
sunmeaserror=[ ];
magmeaserror=[ ];

while T(count)<=total_time

  %************ simulate attitude motion in terms of Euler angles **************
  %*******************************************************************

  %******* Calculate Euler angles and noisy measurements at each time step ********

  %******* Use Euler angles to generate direction cosine matrix at each time *******
  %******* step matrix to convert from inertial to body axes...from [30], *******
  %******* pg 36 or [40], pg 7                                 *******

  DCM(1,1)=cos(phi(count))*cos(psi(count))-sin(phi(count))*cos(theta(count))*sin(psi(count));
  DCM(1,2)=cos(phi(count))*sin(psi(count))+sin(phi(count))*cos(theta(count))*cos(psi(count));
  DCM(1,3)=sin(phi(count))*sin(theta(count));
  DCM(2,1)=-sin(phi(count))*cos(psi(count))-cos(phi(count))*cos(theta(count))*sin(psi(count));
  DCM(2,2)=-sin(phi(count))*sin(psi(count))+cos(phi(count))*cos(theta(count))*cos(psi(count));
  DCM(2,3)=cos(phi(count))*sin(theta(count));
  DCM(3,1)=sin(theta(count))*sin(psi(count));
  DCM(3,2)=-sin(theta(count))*cos(psi(count));
  DCM(3,3)=cos(theta(count));
```

```
%******* Use DCM to rotate inertial reference vectors to body frame *******
%******* Ref vectors are hard-coded or user entered here, but would *******
%******* come from solar ephemeris and mag field model in real life *******
%******* given time and position from radar, gps or other source    *******

SUNVEC(1:3,count)=[SUNX;SUNY;SUNZ];                    %builds array of inertial sun reference vector
if norm(SUNVEC)>eps
   SUNVEC(1:3,count)=SUNVEC(1:3,count)/norm(SUNVEC(1:3,count));   %insures reference vectors are unit vectors
end
MAGVEC(1:3,count)=[MAGX;MAGY;MAGZ];                    %builds array of inertial mag field reference vectors
if norm(MAGVEC)>eps
   MAGVEC(1:3,count)=MAGVEC(1:3,count)/norm(MAGVEC(1:3,count));   %insures reference vectors are unit vectors
end

sunvec(1:3,count)=DCM*SUNVEC(1:3,count);               %rotates SUNVEC to body frame
magvec(1:3,count)=DCM*MAGVEC(1:3,count);                %rotates MAGVEC to body frame

%******* Add random noise with defined sigma to simulate noisy measurements ******

sunmeas(1,count)=sunvec(1,count)+SIGSUNX*randn;   %builds array of simulated noisy sun sensor measurements
sunmeas(2,count)=sunvec(2,count)+SIGSUNY*randn;   %builds array of simulated noisy sun sensor measurements
sunmeas(3,count)=sunvec(3,count)+SIGSUNZ*randn;   %builds array of simulated noisy sun sensor measurements
magmeas(1,count)=magvec(1,count)+SIGMAGX*randn;   %builds array of simulated noisy sun sensor measurements
magmeas(2,count)=magvec(2,count)+SIGMAGY*randn;   %builds array of simulated noisy sun sensor measurements
magmeas(3,count)=magvec(3,count)+SIGMAGZ*randn;   %builds array of simulated noisy sun sensor measurements

%******* Add random noise with defined sigma to simulate noisy measurements ******

omega1meas(1,count)=omega1(1,count)+SIGOMEGA1*randn;   %builds array of simulated noisy sun sensor measurements
omega2meas(1,count)=omega2(1,count)+SIGOMEGA2*randn;   %builds array of simulated noisy sun sensor measurements
omega3meas(1,count)=omega3(1,count)+SIGOMEGA3*randn;   %builds array of simulated noisy sun sensor measurements

%******* Convert Euler angles to the "true" attitude quaternion at each time step *******
%******* This provides "truth" to compare back to at outcome of filtering algorithm *****

q1(count)=cos(psi(count)/2)*sin(theta(count)/2)*cos(phi(count)/2)+sin(psi(count)/2)*sin(theta(count)/2)*sin(phi(count)/2);
q2(count)=-cos(psi(count)/2)*sin(theta(count)/2)*sin(phi(count)/2)+sin(psi(count)/2)*sin(theta(count)/2)*cos(phi(count)/2);
q3(count)=cos(psi(count)/2)*cos(theta(count)/2)*sin(phi(count)/2)+sin(psi(count)/2)*cos(theta(count)/2)*cos(phi(count)/2);
q4(count)=cos(psi(count)/2)*cos(theta(count)/2)*cos(phi(count)/2)-sin(psi(count)/2)*cos(theta(count)/2)*sin(phi(count)/2);

Qvec(1:4,count)=[q1(count); q2(count); q3(count); q4(count)];   %array of rotation quaternions

%*********************** Error arrays for analysis ********************
```

```
 sunmeaserror(1:3,count)=sunvec(1:3,count)-sunmeas(1:3,count);
 magmeaserror(1:3,count)=magvec(1:3,count)-magmeas(1:3,count);
 omega1measerror(1:count)=omega1(1:count)-omega1meas(1:count);
 omega2measerror(1:count)=omega2(1:count)-omega2meas(1:count);
 omega3measerror(1:count)=omega3(1:count)-omega3meas(1:count);

 TI=0.;                                                              %initialize integration timer
 PSI=psi(count);              %initialize psi for numerical integration
 THETA=theta(count);             %initialize theta for numerical integration
 PHI=phi(count);               %initialize phi for numerical integration
 OMEGA1=omega1(count);              %initialize omega1 for numerical integration
 OMEGA2=omega2(count);              %initialize omega2 for numerical integration
 OMEGA3=omega3(count);              %initialize omega3 for numerical integration

 if abs(THETA)<eps
  THETA=1e-4;                  %prevent divide by zero / singularity at theta=0 rad
end
 if abs(THETA-pi)<eps            %prevent divide by zero / singularity at theta=pi rad
   if sign(THETA-pi)==1
     THETA=theta+1e-4;
   else
     THETA=theta-1e-4;
   end
 end
 if abs(THETA-2*pi)<eps            %prevent divide by zero / singularity at theta=pi rad
   if sign(THETA-2*pi)==1
     THETA=theta+1e-4;
   else
     THETA=theta-1e-4;
   end
 end
 while TI<=(TS-.0001)

  %******* propagate Euler angles forward one time step using 2nd order Runge-Kutta *******
  %******* algorithm to numerically integrate the Euler equations                   *******

  PSIOLD=PSI;
  THETAOLD=THETA;
  PHIOLD=PHI;

  PSIDOT=1/sin(THETA)*(sin(PHI)*OMEGA1+cos(PHI)*OMEGA2);
  THETADOT=1/sin(THETA)*(cos(PHI)*sin(THETA)*OMEGA1-sin(PHI)*sin(THETA)*OMEGA2);
  PHIDOT=1/sin(THETA)*(-sin(PHI)*cos(THETA)*OMEGA1-cos(PHI)*cos(THETA)*OMEGA2+sin(THETA)*OMEGA3);
```

```
    PSI=PSI+H*PSIDOT;                                        %calculate psi at next time step
    THETA=THETA+H*THETADOT;                                  %calculate theta at next time step
    PHI=PHI+H*PHIDOT;                                        %calculate phi at next time step

    PSIDOT=1/sin(THETA)*(sin(PHI)*OMEGA1+cos(PHI)*OMEGA2);
    THETADOT=1/sin(THETA)*(cos(PHI)*sin(THETA)*OMEGA1-sin(PHI)*sin(THETA)*OMEGA2);
    PHIDOT=1/sin(THETA)*(-sin(PHI)*cos(THETA)*OMEGA1-cos(PHI)*cos(THETA)*OMEGA2+sin(THETA)*OMEGA3);

    PSI=.5*(PSIOLD+PSI+H*PSIDOT);                             %num integrate psi forward
    THETA=.5*(THETAOLD+THETA+H*THETADOT);                    %num integrate theta forward
    PHI=.5*(PHIOLD+PHI+H*PHIDOT);                            %num integrate phi forward

    TI=TI+H;                                                 %increment integration time
  end

  count=count+1;                                             %increment counter
  T(1,count)=T(1,count-1)+TS;                                %increment simulation time



  %****** store quantities of interest in arrays ******

  psidot(count)=PSIDOT;
  thetadot(count)=THETADOT;
  phidot(count)=PHIDOT;
  psi(count)=PSI;
  theta(count)=THETA;
  phi(count)=PHI;

end

%******** represent psi, theta, phi as angles between 0 and 2pi *******

for j=1:count
  psi(j)=mod(psi(j),2*pi);                    %represent psi between 0 and 2pi
  theta(j)=mod(theta(j),2*pi);                %represent theta between 0 and 2pi
  phi(j)=mod(phi(j),2*pi);                    %represent phi between 0 and 2pi
end

%******** Adjust count to match array sizes for error calculation and plotting ******
%*** ...necessary because count gets incremented before failing test to continue loop ***

count=count-1;
```

```
%******** truncate data arrays to correct length**********
%*** ...necessary because T gets incremented before failing test to continue loop ***

T=T(1:count);

%******* Option to simulate "drop-outs" in measurements *******

'Please press "return" to continue...'
pause;close;clc;

'Simulation data generation (continued)...'
''
'Do you wish to simulate data dropouts? (default is no)'
dropout=input('Please enter "1" to simulate dropouts or "0" for no dropouts...(default is no dropouts): ');
if isempty(dropout)
  dropout=0
end

if dropout == 1
  ''
  dropsunmeas=input('enter duration of sunsensor dropout in seconds: ');
  dropsunmeas=floor(dropsunmeas/TS);
  dropsunwhere=input('enter where in simulated data drop occurs as a decimal (i.e., half way = .5): ');
  ''
  dropmagmeas=input('enter duration of magsensor dropout in seconds: ');
  dropmagmeas=floor(dropmagmeas/TS);
  dropmagwhere=input('enter where in simulated data drop occurs as a decimal (i.e., half way = .5): ');
  ''
  sunstartcount=floor(dropsunwhere*length(sunmeas));
  sunmeas(1:3,sunstartcount:sunstartcount+dropsunmeas)=eps;

  magstartcount=floor(dropmagwhere*length(magmeas));
  magmeas(1:3,magstartcount:magstartcount+dropmagmeas)=eps;
  ''
  dropomegameas=input('enter duration of rate sensor dropout in seconds: ');
  dropomegameas=floor(dropomegameas/TS);
  dropomegawhere=input('enter where in simulated data drop occurs as a decimal (i.e., half way = .5): ');
  ''
  omegastartcount=floor(dropomegawhere*length(omega1meas));
  omega1meas(1,omegastartcount:omegastartcount+dropomegameas)=eps;
  omega2meas(1,omegastartcount:omegastartcount+dropomegameas)=eps;
  omega3meas(1,omegastartcount:omegastartcount+dropomegameas)=eps;
end
```

```
%*********************** Data Review and Plotting *********************

plots=[ ];
plots=input('Enter "1" to show plots, "0" to skip plots. Default is to show plots:  ');
if isempty(plots)
  plots=1;
end

if plots == 1

  %*************** sun vector plots ********************

  plot(T,sunmeas(1,1:count),'g+', T, sunvec(1,1:count), 'r', T,SUNVEC(1,1:count),'o'),grid on;
  title('X component of unit sun vector vs. time');xlabel('time (sec)');ylabel('X component');
  legend('measurement', 'rotated', 'inertial');
  pause;close;clc;

  plot(T,sunmeas(2,1:count),'g+', T, sunvec(2,1:count), 'r',T,SUNVEC(2,1:count),'o'),grid on;
  title('Y component of unit sun vector vs. time');xlabel('time (sec)');ylabel('Y component');
  legend('measurement', 'rotated', 'inertial');
  pause;close;clc;

  plot(T,sunmeas(3,1:count),'g+', T, sunvec(3,1:count),'r',T,SUNVEC(3,1:count),'o'),grid on;
  title('Z component of unit sun vector vs. time');xlabel('time (sec)');ylabel('Z component');
  legend('measurement', 'rotated', 'inertial');
  pause;close;clc;

  plot3(sunvec(1,1:count),sunvec(2,1:count),sunvec(3,1:count),'r+');grid on;hold on;
  plot3(SUNVEC(1,1:count),SUNVEC(2,1:count),SUNVEC(3,1:count),'b*');
  axis([min(sunvec(1,1:count))-1,max(sunvec(1,1:count))+1,min(sunvec(2,1:count))-
1,max(sunvec(2,1:count))+1,min(sunvec(3,1:count))-1,max(sunvec(3,1:count))+1]);
  xlabel('X component');ylabel('Y component');zlabel('Z component');
  legend('rotated','inertial');
  title('3-D plot of both inertial and rotated sun vector from start to end of simulation');
  pause;close;clc;

  %*************** mag field vector plots ********************

  plot(T,magmeas(1,1:count),'g+', T, magvec(1,1:count), 'r',T,MAGVEC(1,1:count),'o');grid on;
  title('X component of unit mag field vector vs. time');xlabel('time (sec)');ylabel('X component');
  legend('measurement', 'rotated', 'inertial');
  pause;close;clc;
```

```
plot(T,magmeas(2,1:count),'g+', T, magvec(2,1:count), 'r',T,MAGVEC(2,1:count),'o');grid on;
title('Y component of unit mag field vector vs. time');xlabel('time (sec)');ylabel('Y component');
legend('measurement', 'rotated', 'inertial');
pause;close;clc;

plot(T,magmeas(3,1:count),'g+', T, magvec(3,1:count), 'r',T,MAGVEC(3,1:count),'o');grid on;
title('Z component of unit mag field vector vs. time');xlabel('time (sec)');ylabel('Z component');
legend('measurement', 'rotated', 'inertial');
pause;close;clc;

plot3(magvec(1,1:count),magvec(2,1:count),magvec(3,1:count),'m+');grid on;hold on;
plot3(MAGVEC(1,1:count),MAGVEC(2,1:count),MAGVEC(3,1:count),'b*');
axis([min(magvec(1,1:count))-1,max(magvec(1,1:count))+1,min(magvec(2,1:count))-
1,max(magvec(2,1:count))+1,min(magvec(3,1:count))-1,max(magvec(3,1:count))+1]);
xlabel('X component');ylabel('Y component');zlabel('Z component');
legend('rotated','inertial');
title('3-D plot of both inertial and rotated mag field vector from start to end of simulation');
pause;close;clc;

%****** plot inertial and rotated frame sun and mag field vectors ******

plot3(sunvec(1,1:count),sunvec(2,1:count),sunvec(3,1:count),'r+');grid on;hold on;
plot3(SUNVEC(1,1:count),SUNVEC(2,1:count),SUNVEC(3,1:count),'bd');
plot3(magvec(1,1:count),magvec(2,1:count),magvec(3,1:count),'mx');
plot3(MAGVEC(1,1:count),MAGVEC(2,1:count),MAGVEC(3,1:count),'ks');
plot3(q1(1:count),q2(1:count),q3(1:count),'go');
xlabel('X component');ylabel('Y component');zlabel('Z component');
legend('rotated sun','inertial sun','rotated mag','inertial mag','quaternion');
title('3-D plot of inertial and rotated sun and mag field vectors and rotation quaternion');
pause;close;clc;

%*************** body frame omega plots ********************

plot(T,omega1meas(1,1:count)*60/(2*pi),'g+', T, omega1(1,1:count)*60/(2*pi), 'r');grid on;
title('rotational rate about body axis 1 vs. time');xlabel('time (sec)');ylabel('omega_1 (rpm)');
legend('measurement', 'actual');
pause;close;clc;

plot(T,omega2meas(1,1:count)*60/(2*pi),'g+', T, omega2(1,1:count)*60/(2*pi), 'r');grid on;
title('rotational rate about body axis 2 vs. time');xlabel('time (sec)');ylabel('omega_2 (rpm)');
legend('measurement', 'actual');
pause;close;clc;

plot(T,omega3meas(1,1:count)*60/(2*pi),'g+', T, omega3(1,1:count)*60/(2*pi), 'r');grid on;
```

```
  title('rotational rate about body axis 3 vs. time');xlabel('time (sec)');ylabel('omega_3 (rpm)');
  legend('measurement', 'actual');
  pause;close;clc;

  %****** plot Euler angle rates ******

  psidotrpm=psidot*60/(2*pi);            %convert angular rates from rad/s to rev/min for plotting
  thetadotrpm=thetadot*60/(2*pi);
  phidotrpm=phidot*60/(2*pi);

  plot (T(5:count),psidotrpm(5:count),'bo-',T(5:count),thetadotrpm(5:count),'rx-',T(5:count),phidotrpm(5:count),'m*-');
  grid on;title('Euler angle rates versus time (without initial transients)');
  xlabel('time (sec)');ylabel('Euler angle rates (rev/min)');legend('psidot', 'thetadot', 'phidot');
  pause;close;

  %**************** Euler angle plots ********************

  degpsi=psi*180/pi;                  %convert psi to degrees for plotting
  degtheta=theta*180/pi;               %convet theta to degrees for plotting
  degphi=phi*180/pi;                  %convert phi to degrees for plotting


  plot(T,degpsi(1:count),'mx',T,degtheta(1:count),'b+',T,degphi(1:count),'r-'),grid on;
  title('Euler angles vs. time');xlabel('time (sec)');ylabel('Euler angles');
  legend('psi (deg)','theta (deg)','phi (deg)');
  pause;close;clc;

  %**************** Attitude quaternion component plots ********************

  plot(T,q4(1:count),'mo',T,q1(1:count),'bx',T,q2(1:count),'r+',T,q3(1:count),'g*'),grid on;
  title('attitude quaternion components vs. time');xlabel('time (sec)');ylabel('quaternion components');
  legend( 'q4', 'q1', 'q2', 'q3');
  pause;close;clc;
end

%****** Diagnostics ******
diagnostics=[ ];
diagnostics=input('Enter "1" to display diagnostic values, "0" to skip diagnostics. Default is no diagnostics:  ');
if isempty(diagnostics)
  diagnostics=0;
end

if diagnostics == 1
```

```
q_rot=Qvec(1:4,count);

Qvec1=q_rot(1);
Qvec2=q_rot(2);
Qvec3=q_rot(3);
Qvec4=q_rot(4);

[axis,angle]=qrot(q_rot)

'press "return" to continue...'
pause;close;clc;

eul(1)=psi(count)*180/pi;
eul(2)=theta(count)*180/pi;
eul(3)=phi(count)*180/pi;

'The final Euler angles are:'
'psi= ',eul(1)
'theta= ',eul(2)
'phi= ',eul(3)

'The final rotated vectors are: '
'sunvec= ',sun=sunvec(1:3,count)
'magvec= ',mag=magvec(1:3,count)

a(1,1)=Qvec1^2-Qvec2^2-Qvec3^2+Qvec4^2;
a(1,2)=2*(Qvec1*Qvec2+Qvec3*Qvec4);
a(1,3)=2*(Qvec1*Qvec2-Qvec3*Qvec4);
a(2,2)=-Qvec1^2+Qvec2^2-Qvec3^2+Qvec4^2;
a(2,3)=2*(Qvec1*Qvec3+Qvec2*Qvec4);
a(3,2)=2*(-Qvec1*Qvec4+Qvec2*Qvec3);
a(3,3)=-Qvec1^2-Qvec2^2+Qvec3^2+Qvec4^2;

vinit=[SUNVEC(1:3,count);MAGVEC(1:3,count)];
vfinal=[sunvec(1:3,count);magvec(1:3,count)];

SUN=SUNVEC(1:3,1);
MAG=MAGVEC(1:3,1);

'press "return" to continue...'
pause;close;clc;

'The standard dev of the generated sun vector x components is:',std(sunmeaserror(1,1:count))
'The standard dev of the generated sun vector y components is:',std(sunmeaserror(2,1:count))
```

```
  'The standard dev of the generated sun vector z components is:',std(sunmeaserror(3,1:count))
  'The standard dev of the generated mag field vector x components is:',std(magmeaserror(1,1:count))
  'The standard dev of the generated mag field vector y components is:',std(magmeaserror(2,1:count))
  'The standard dev of the generated mag field vector z components is:',std(magmeaserror(3,1:count))
  'The standard dev of the generated omega1 measurement is:',std(omega1measerror(1:count))
  'The standard dev of the generated omega2 measurement is:',std(omega2measerror(1:count))
  'The standard dev of the generated omega3 measurement is:',std(omega3measerror(1:count))

end

' '
' '
'Data simulation complete...press "return" to begin processing data'
pause;close;clc;
```

A.3 Omegagen_exp.m

Omegagen_exp.m is a user defined function that is called during the execution of attitude_sim.m, if designated by the operator. This routine generates body frame rotational rate profiles that asymptotically approach a final value, input by the user for each axis.

```
%Omegagen_exp - user defined function that generates omega arrays
%for use by attitude_sim. User defined omega profiles that
%asymptotically approach the final desired RPM at user-defined times.
%User must input initial omega, ramp variables and desired final rpm.
%
%function [omega1, omega2, omega3,TS]=omegagen_exp(total_time)
%
%omega1 = angular rate about body frame x-axis
%omega2 = angular rate about body frame y-axis
%omega3 = angular rate about body frame z-axis
%total_time = length of simulation
%
%Written by Mark Charlton. Last updated 27 October 03.


function [omega1, omega2, omega3, TS]=omegagen_exp(total_time)

clc

%initialize arrays
```

```
omega1=[ ];
omega2=[ ];
omega3=[ ];

%***************************************************************************
%****** generate arrays of angular rates about the rotated frame axes                    ******
%***************************************************************************

%Input rate profile information about body frame x-axis, omega1

omega1=input('Please enter the initial value of the angular rate about the body frame x axis in rpm (default = 0):  ');
if isempty(omega1)
  omega1=0
end

omega1final=input('Please enter the final value of the angular rate about the body frame x axis in rpm (default = .5):  ');
if isempty(omega1final)
  omega1final=.5
end

omega1ramp=input('Please enter the fraction of the entire simulation at which you which final value of the angular rate to be
achieved (default = 1/2):  ');
if isempty(omega1ramp)
  omega1ramp=.5
end

omega2=input('Please enter the initial value of the angular rate about the body frame y axis in rpm (default = 0):  ');
if isempty(omega2)
  omega2=0
end

omega2final=input('Please enter the final value of the angular rate about the body frame y axis in rpm (default = .5):  ');
if isempty(omega2final)
  omega2final=.5
end

omega2ramp=input('Please enter the fraction of the entire simulation at which you wish final value of the angular rate to be achieved
(default = 1/2):  ');
if isempty(omega2ramp)
  omega2ramp=.5
end

omega3=input('Please enter the initial value of the angular rate about the body frame z axis in rpm (default = 0):  ');
if isempty(omega3)
```

```
  omega3=0
end

omega3final=input('Please enter the final value of the angular rate about the body frame z axis in rpm (default = 225):  ');
if isempty(omega3final)
  omega3final=225
end

omega3ramp=input('Please enter the fraction of the entire simulation at which you wish final value of the angular rate to be achieved
(default = 1/2):  ');
if isempty(omega3ramp)
  omega3ramp=.5
end

%******* determine highest frequency of simulated motion *******

highestfreq=max([omega1final omega2final omega3final])*2*pi/60;

%******* simulation time step **************

'***** Care should be taken to choose a simulation/sampling interval at least 3 times highest frequency of rotational motion!!!*****'
TS=input('Please enter the attitude parameter simulation interval in seconds (default = .01 (100 s/sec)):  ');

if isempty(TS)
  if highestfreq==0
    TS=1
  else
    TS=.01
  end
end

%******* generate time array based on final time and time step *******

T=0:TS:total_time;
count=length(T);

omega1(1,1)=omega1;
omega2(1,1)=omega2;
omega3(1,1)=omega3;


ramp1=floor(count*omega1ramp);
ramp2=floor(count*omega2ramp);
ramp3=floor(count*omega3ramp);
```

```
for i=2:count
   omega1(1,i)=(omega1final*2*pi/60)*(1-exp(-(5/ramp1)*i));   %generate omega1 profile
end
for j=2:count
   omega2(1,j)=(omega2final*2*pi/60)*(1-exp(-(5/ramp2)*j));   %generate omega2 profile
end
for k=2:count
   omega3(1,k)=(omega3final*2*pi/60)*(1-exp(-(5/ramp3)*k));   %generate omega3 profile
end

   %****** plot angular rates in the body frame ******

   %plot
(T(1,1:count),omega1(1,1:count)*60/(2*pi),'bo',T(1,1:count),omega2(1,1:count)*60/(2*pi),'rx',T(1,1:count),omega3(1,1:count)*60/(2
*pi),'m*');
   %grid on;title('body rates versus time');
   %xlabel('time (sec)');ylabel('body rates (rev/min)');legend('omega1', 'omega2', 'omega3');
   %pause;close;

%end of function
```

A.4 Omegagen_ramp.m

Omegagen_ramp.m is a user defined function that is called during the execution of attitude_sim.m, if designated by the operator. This routine generates body frame rotational rate profiles that linearly approach a final value, input by the user for each axis.

```
%omegagen_ramp - user defined function that generates omega arrays
%for use by attitude_sim. User defined omega profiles that have
%a linear ramp rise reaching the final desired RPM at defined times.
%User must input initial omega, ramp variables and desired final rpm.
%
%function [omega1, omega2, omega3,TS]=omegagen_ramp(total_time)
%
%omega1 = angular rate about body frame x-axis
%omega2 = angular rate about body frame y-axis
%omega3 = angular rate about body frame z-axis
%total_time = length of simulation
%
%Written by Mark Charlton. Last updated 21 October 03.
```

```
function [omega1, omega2, omega3,TS]=omegagen_ramp(total_time)

clc

%initialize arrays

omega1=[ ];
omega2=[ ];
omega3=[ ];

%****************************************************************************
%****** generate arrays of angular rates about the rotated frame axes ******
%****************************************************************************

%Input rate profile information about body frame x-axis, omega1

omega1=input('Please enter the initial value of the angular rate about the body frame x axis in rpm (default = 0):  ');
if isempty(omega1)
  omega1=0
end

omega1final=input('Please enter the final value of the angular rate about the body frame x axis in rpm (default = .5):  ');
if isempty(omega1final)
  omega1final=.5
end

omega1ramp=input('Please enter the fraction of the entire simulation at which you wish final value of the angular rate to be achieved (default = 1/2:  ');
if isempty(omega1ramp)
  omega1ramp=.5
end

omega2=input('Please enter the initial value of the angular rate about the body frame y axis in rpm (default = 0):  ');
if isempty(omega2)
  omega2=0
end

omega2final=input('Please enter the final value of the angular rate about the body frame y axis in rpm (default = .5):  ');
if isempty(omega2final)
  omega2final=.5
end

omega2ramp=input('Please enter the fraction of the entire simulation at which you wish final value of the angular rate to be achieved (default = 1/2):  ');
```

```
if isempty(omega2ramp)
  omega2ramp=.5
end

omega3=input('Please enter the initial value of the angular rate about the body frame z axis in rpm (default = 0):  ');
if isempty(omega3)
  omega3=0
end

omega3final=input('Please enter the final value of the angular rate about the body frame z axis in rpm (default = 225):  ');
if isempty(omega3final)
  omega3final=225
end

omega3ramp=input('Please enter the fraction of the entire simulation at which you wish final value of the angular rate to be achieved
(default = 1/2):  ');
if isempty(omega3ramp)
  omega3ramp=.5
end

%******* determine highest frequency of simulated motion *******

highestfreq=max([omega1final omega2final omega3final])*2*pi/60;

%******* simulation time step **************

'***** Care should be taken to choose a simulation/sampling interval at least 3 times highest frequency of rotational motion!!!*****'
TS=input('Please enter the attitude parameter simulation interval in seconds (default = .01 (100 s/sec)):  ');

if isempty(TS)
  if highestfreq==0
     TS=1
  else
     TS=.01
  end
end

%******* generate time array based on final time and time step *******

T=0:TS:total_time;
count=length(T);

omega1(1,1)=omega1;
omega2(1,1)=omega2;
```

```
omega3(1,1)=omega3;

ramp1=floor(count*omega1ramp);
ramp2=floor(count*omega2ramp);
ramp3=floor(count*omega3ramp);

for i=2:ramp1+1
   omega1(1,i)=omega1(1,i-1)+(omega1final*2*pi/60)/ramp1;    %generate omega1 profile
end
for i=ramp1+2:count
   omega1(1,i)=omega1(1,i-1);
end
for j=2:ramp2+1
   omega2(1,j)=omega2(1,j-1)+(omega2final*2*pi/60)/ramp2;    %generate omega2 profile
end
for j=ramp2+2:count
   omega2(1,j)=omega2(1,j-1);
end
for k=2:ramp3+1
   omega3(1,k)=omega3(1,k-1)+(omega3final*2*pi/60)/ramp3;    %generate omega3 profile
end
for k=ramp3+2:count
   omega3(1,k)=omega3(1,k-1);
end

   %****** plot angular rates in the body frame ******

   %plot
(T(1,1:count),omega1(1,1:count)*60/(2*pi),'bo',T(1,1:count),omega2(1,1:count)*60/(2*pi),'rx',T(1,1:count),omega3(1,1:count)*60/(2
*pi),'m*');
   %grid on;title('body rates versus time');
   %xlabel('time (sec)');ylabel('body rates (rev/min)');legend('omega1', 'omega2', 'omega3');
   %pause;close;

%end of function
```

A.5 Omegagen_fix.m

Omegagen_fix.m is a user defined function that is called during the execution of attitude_sim.m, if designated by the operator. This routine generates body frame rotational rate profiles that have a fixed value, input by the user for each axis.

```
%omegagen_fix - user defined function that generates omega arrays
```

```
%for use by attitude_sim. User defined omega profiles that have
%a fixed angular rates. User must input desired fixed rpm.
%
%function [omega1, omega2, omega3,TS]=omegagen_ramp(total_time)
%
%omega1 = angular rate about body frame x-axis
%omega2 = angular rate about body frame y-axis
%omega3 = angular rate about body frame z-axis
%total_time = length of simulation
%
%Written by Mark Charlton. Last updated 21 October 03.


function [omega1, omega2, omega3,TS]=omegagen_fix(total_time)

clc

%initialize arrays

omega1=[ ];
omega2=[ ];
omega3=[ ];

%*****************************************************************************
%****** generate arrays of angular rates about the rotated frame axes ******
%*****************************************************************************

%Input rate profile information about body frame x-axis, omega1

omega1=input('Please enter the desired value of the constant angular rate about the body frame x axis in rpm (default = .5): ');
if isempty(omega1)
  omega1=.5
end

omega1=omega1*2*pi/60;

omega2=input('Please enter the desired value of the constant angular rate about the body frame y axis in rpm (default = .5): ');
if isempty(omega2)
  omega2=.5
end

omega2=omega2*2*pi/60;

omega3=input('Please enter the desired value of the constant angular rate about the body frame z axis in rpm (default = 225): ');
```

```
if isempty(omega3)
  omega3=225
end

omega3=omega3*2*pi/60;

%******* determine highest frequency of simulated motion *******

highestfreq=max([omega1 omega2 omega3]);

%******* simulation time step **************

'***** Care should be taken to choose a simulation/sampling interval at least 3 times highest frequency of rotational motion!!!*****'
TS=input('Please enter the attitude parameter simulation interval in seconds (default = .01 (100 s/sec)):  ');

if isempty(TS)
  if highestfreq==0
     TS=1
  else
     TS=.01
  end
end

%******* generate time array based on final time and time step *******

T=0:TS:total_time;
count=length(T);

omega1(1,1)=omega1;
omega2(1,1)=omega2;
omega3(1,1)=omega3;

for i=2:count
   omega1(1,i)=omega1(1,i-1);                    %generate omega1 profile
end

for j=2:count
   omega2(1,j)=omega2(1,j-1);                    %generate omega2 profile
end

end
for k=2:count
   omega3(1,k)=omega3(1,k-1);                     %generate omega3 profile
end
```

```
  %****** plot angular rates in the body frame ******

  %plot
(T(1,1:count),omega1(1,1:count)*60/(2*pi),'bo',T(1,1:count),omega2(1,1:count)*60/(2*pi),'rx',T(1,1:count),omega3(1,1:count)*60/(2
*pi),'m*');
  %grid on;title('body rates versus time');
  %xlabel('time (sec)');ylabel('body rates (rev/min)');legend('omega1', 'omega2', 'omega3');
  %pause;close;

%end of function
```

A.6 Omegagen_step.m

Omegagen_step.m is a user defined function that is called during the execution of attitude_sim.m, if designated by the operator. This routine generates body frame rotational rate profiles that have a step shape with start and stop times and maximum values input by the user for each axis.

```
%omegagen_step - user defined function that generates omega arrays
%for use by attitude_sim. User defined omega profiles that have
%a step rise to the final desired RPM at a defined time and a step drop
%back to zero at another defined time.
%User must input initial omega, ramp variables and desired non-zero rpm.
%
%function [omega1, omega2, omega3,TS]=omegagen_step(total_time)
%
%omega1 = angular rate about body frame x-axis
%omega2 = angular rate about body frame y-axis
%omega3 = angular rate about body frame z-axis
%total_time = length of simulation
%
%Written by Mark Charlton. Last updated 21 October 03.


function [omega1, omega2, omega3,TS]=omegagen_step(total_time)

clc

%initialize arrays

omega1=[ ];
omega2=[ ];
```

```
omega3=[ ];

%**************************************************************************
%****** generate arrays of angular rates about the rotated frame axes ******
%**************************************************************************

%Input rate profile information about body frame x-axis, omega1

omega1=input('Please enter the initial value of the angular rate about the body frame x axis in rpm (default = 0):  ');
if isempty(omega1)
  omega1=0
end
omega1=omega1*2*pi/60;          %convert to rad/s

omega1final=input('Please enter the final value of the angular rate about the body frame x axis in rpm (default = .2):  ');
if isempty(omega1final)
  omega1final=.2
end

omega1final=omega1final*2*pi/60;    %convert to rad/s

omega1stepup=input('Please enter the fraction of the entire simulation at which you wish pulse to begin (default = .3):  ');
if isempty(omega1stepup)
  omega1stepup=3/10
end

omega1stepdown=input('Please enter the fraction of the entire simulation at which you wish pulse to end (default = .7):  ');
if isempty(omega1stepdown)
  omega1stepdown=7/10
end

%Input rate profile information about body frame y-axis, omega2

omega2=input('Please enter the initial value of the angular rate about the body frame y axis in rpm (default = 0):  ');
if isempty(omega2)
  omega2=0
end

omega2=omega2*2*pi/60;          %convert to rad/s

omega2final=input('Please enter the final value of the angular rate about the body frame y axis in rpm (default = .2):  ');
if isempty(omega2final)
  omega2final=.2
end
```

```
omega2final=omega2final*2*pi/60;    %convert to rad/s

omega2stepup=input('Please enter the fraction of the entire simulation at which you wish pulse to begin (default = .3):  ');
if isempty(omega2stepup)
  omega2stepup=3/10
end

omega2stepdown=input('Please enter the fraction of the entire simulation at which you wish pulse to end (default = .7):  ');
if isempty(omega2stepdown)
  omega2stepdown=7/10
end

%Input rate profile information about body frame z-axis, omega3

omega3=input('Please enter the initial value of the angular rate about the body frame z axis in rpm (default = 0):  ');
if isempty(omega3)
  omega3=0
end

omega3=omega3*2*pi/60;              %convert to rad/s

omega3final=input('Please enter the final value of the angular rate about the body frame z axis in rpm (default = .2):  ');
if isempty(omega3final)
  omega3final=.2
end

omega3final=omega3final*2*pi/60;    %convert to rad/s

omega3stepup=input('Please enter the fraction of the entire simulation at which you wish pulse to begin (default = .3):  ');
if isempty(omega3stepup)
  omega3stepup=3/10
end

omega3stepdown=input('Please enter the fraction of the entire simulation at which you wish pulse to end (default = .7):  ');
if isempty(omega3stepdown)
  omega3stepdown=7/10
end

%******* determine highest frequency of simulated motion *******

highestfreq=max([omega1final omega2final omega3final])*2*pi/60;

%******* simulation time step **************
```

```
'***** Care should be taken to choose a simulation/sampling interval at least 3 times highest frequency of rotational motion!!!*****'
TS=input('Please enter the attitude parameter simulation interval in seconds (default = .01 (100 s/sec)):  ');
if isempty(TS)
  TS=0.01
end

%******* generate time array based on final time and time step *******

T=0:TS:total_time;
count=length(T);

omega1(1,1)=omega1;
omega2(1,1)=omega2;
omega3(1,1)=omega3;

stepup1=floor(count*omega1stepup);
stepdown1=floor(count*omega1stepdown);
stepup2=floor(count*omega2stepup);
stepdown2=floor(count*omega2stepdown);
stepup3=floor(count*omega3stepup);
stepdown3=floor(count*omega3stepdown);

for i=2:stepup1
   omega1(1,i)=omega1(1,1);     %generate omega1 profile
end
for i=stepup1+1:stepdown1
   omega1(1,i)=omega1final;
end
for i=stepdown1+1:count;
   omega1(1,i)=omega1(1,1);
end

for j=2:stepup2
   omega2(1,j)=omega2(1,1);     %generate omega2 profile
end
for j=stepup2+1:stepdown2
   omega2(1,j)=omega2final;
end
for j=stepdown2+1:count;
   omega2(1,j)=omega2(1,1);
end

for k=2:stepup3
```

```
  omega3(1,k)=omega3(1,1);    %generate omega3 profile
end
for k=stepup3+1:stepdown3
  omega3(1,k)=omega3final;
end
for k=stepdown3+1:count;
  omega3(1,k)=omega3(1,1);
end


  %****** plot angular rates in the body frame ******

  %plot
(T(1,1:count),omega1(1,1:count)*60/(2*pi),'bo',T(1,1:count),omega2(1,1:count)*60/(2*pi),'rx',T(1,1:count),omega3(1,1:count)*60/(2
*pi),'m*');
  %grid on;title('body rates versus time');
  %xlabel('time (sec)');ylabel('body rates (rev/min)');legend('omega1', 'omega2', 'omega3');
  %pause;close;

%end of function
```

A.7 Qrot.m

This is a user defined function that calculates the familiar Euler rotation axis and angle representation of a rotation from a given rotation quaternion. It is called from attitude_sim.m if the "diagnostics" option is selected.

```
%qrot - user defined function to calculate rotation and axis corresponding
%to a given rotation quaternion.
%
%function [axis,angle]=qrot(q_rot)
%
%q_rot = rotation quaternion of unit magnitude
%axis = Euler axis of rotation
%angle = angle of rotation about axis
%
%Written by Mark Charlton. Last updated 28 October 2003.

function [axis,angle]=qrot(q_rot)
clc;
'The entered rotation quaternion is: '
q_rot
```

```
q_rot=q_rot/norm(q_rot);        %insures unit quaternion

%Calculate rotation generated by q_rot

angle=2*acos(q_rot(4));

angle=angle*180/pi;             %convert angle to degrees
''
axis=[q_rot(1) q_rot(2) q_rot(3)]';
axis=axis/sqrt(axis'*axis);

'The unit vector representing the Euler axis of rotation and the angle of rotation in degrees are: '

%end of function
```

A.8 Gauss_Newton.m

Gauss_Newton.m is a user defined MATLAB function to implement Gauss-Newton error minimization for a quadratic error function. This function is called by the main script attitude_filter.m and operates on the noisy measurements coming out of attitude_sim.m. It produces the "best" quaternion estimate from the error minimization process. These noisy quaternion components are then operated on by the chosen EKF or UKF filter which produces the state estimate.

```
%Gauss_Newton - user defined function to minimize errors in transformation between two given vectors.
%Uses Gauss-Newton algorithm to minimize mean-square-error (MSE) [65 ]. The error function is
%of the form (vref-M*vrot)'*(vref-M*vrot). Function yields rotation quaternion that describes
%transformation between vector in rotated frame and vector in reference frame that yields lowest MSE.
%
%function [q_hat, A, check,err]=Gauss_newton(vref, vrot, q_init,steps,tol)
%
%q_hat = estimated rotation quaternion based on minimized MSE
%
%A = matrix that calculates measurement errors of the four computed
%quaternion elements based on the measurement errors of the six sensor
%measurements [11]
%
%check = convergence flag for algorithm. check==2 indicates normal
%convergence. check==0 indicates that algorithm did not converge in
%"steps" iterations.
%
%vref = vector in reference frame (6 x 1)
%vrot = vector in rotated frame (6 x 1)
```

```
%q_init = initial guess at unit rotation quaternion (4 x 1)
%steps = max steps to converge
%tol = convergence tolerance
%
%Written by Mark Charlton. Last updated 21 October 03.

function [q_hat, A, check, err] = Gauss_newton(vref, vrot, q_init, steps, tol)

%Normalize initial quaternion guess to get required unit vector

q_init=q_init/sqrt(q_init'*q_init);

%Initialize variables

q_hat=q_init;     %rotation quaternion resulting from minimizing MSE
A=zeros(4,6);     %matrix relating quaternion error to measurement error
M_hat=zeros(6,6); %transformation matrix in error function
err=1;            %error used for convergence criteria
i=0;              %counter to prevent infinite loop
check=1;          %flag to indicate status of convergence
q_hat1=[q_init(1)];
q_hat2=[q_init(2)];
q_hat3=[q_init(3)];
q_hat4=[q_init(4)];
steparray=[i];
errarray=[err];

%Iterate to minimize MSE and yield optimum rotation quaternion

while check==1

  %Quaternion rotation matrix describing transformation from body frame to
  %reference frame [40]

  R_hat(1,1)=q_hat(1)^2-q_hat(2)^2-q_hat(3)^2+q_hat(4)^2;
  R_hat(1,2)=2*(q_hat(1)*q_hat(2)-q_hat(3)*q_hat(4));
  R_hat(1,3)=2*(q_hat(1)*q_hat(3)+q_hat(2)*q_hat(4));
  R_hat(2,1)=2*(q_hat(1)*q_hat(2)+q_hat(3)*q_hat(4));
  R_hat(2,2)=-q_hat(1)^2+q_hat(2)^2-q_hat(3)^2+q_hat(4)^2;
  R_hat(2,3)=2*(-q_hat(1)*q_hat(4)+q_hat(2)*q_hat(3));
  R_hat(3,1)=2*(q_hat(1)*q_hat(3)-q_hat(2)*q_hat(4));
  R_hat(3,2)=2*(q_hat(1)*q_hat(4)+q_hat(2)*q_hat(3));
  R_hat(3,3)=-q_hat(1)^2-q_hat(2)^2+q_hat(3)^2+q_hat(4)^2;
```

```
%Form matrix M in error vector errvec=(vref-Mvrot)

M_hat=[R_hat, zeros(3,3); zeros(3,3), R_hat];

%Calculate partial derivatives to form Jacobian matrix, jacobian

subq1=[q_hat(1) q_hat(2) q_hat(3); q_hat(2) -q_hat(1) -q_hat(4); q_hat(3) q_hat(4) -q_hat(1)];
partq1=2*[subq1, zeros(3,3); zeros(3,3), subq1]*vrot;

subq2=[-q_hat(2) q_hat(1) q_hat(4); q_hat(1) q_hat(2) q_hat(3); -q_hat(4) q_hat(3) -q_hat(2)];
partq2=2*[subq2, zeros(3,3); zeros(3,3), subq2]*vrot;

subq3=[-q_hat(3) -q_hat(4) q_hat(1); q_hat(4) -q_hat(3) q_hat(2); q_hat(1) q_hat(2) q_hat(3)];
partq3=2*[subq3, zeros(3,3); zeros(3,3), subq3]*vrot;

subq4=[q_hat(4) -q_hat(3) q_hat(2); q_hat(3) q_hat(4) -q_hat(1); -q_hat(2) q_hat(1) q_hat(4)];
partq4=2*[subq4, zeros(3,3); zeros(3,3), subq4]*vrot;

%Calculate Jacobian matrix, jacobian

jacobian=-[partq1, partq2, partq3, partq4];

%Calculate new estimate of quaternion

q_hat=q_hat-(inv(jacobian'*jacobian))*jacobian'*(vref-M_hat*vrot);

%Normalize estimated quaternion to get required unit quaternion

q_hat=q_hat/sqrt(q_hat'*q_hat);

%Check error vector

errvec=vref-M_hat*vrot;
err=(errvec'*errvec)/length(errvec);    %calculate mean square error

%increment counter

i=i+1;

%Check for convergence of Gauss-Newton

if i >= steps
   check = 0;   %set convergence flag
   %'Gauss-Newton failed to converge in 25 steps...'
```

```
    %pause(1);
    %err
  end

  if err < tol
    check = 2;  %set convergence flag
    %'Gauss-Newton has converged...'
    %pause(3);
  end

  q_hat1=[q_hat1, q_hat(1)];     %array containing q_hat(1) at each time step
  q_hat2=[q_hat2, q_hat(2)];     %array containing q_hat(2) at each time step
  q_hat3=[q_hat3, q_hat(3)];     %array containing q_hat(3) at each time step
  q_hat4=[q_hat4, q_hat(4)];     %array containing q_hat(4) at each time step
  errarray=[errarray, err];      %array containign err at each time step
  steparray=[steparray, i];      %array containing all time steps

end

%Calculate measurement errors of four computed quaternion elements
%based on the measurement errors of the six sensor measurements [11]

A=(inv(jacobian'*jacobian))*jacobian'*M_hat;

%Normalize estimated quaternion to get required unit quaternion

q_hat=q_hat/sqrt(q_hat'*q_hat);

%diagnostic plots of quaternion components

%plot(steparray,q_hat1,steparray,q_hat2,'-r', steparray, q_hat3, '-g',steparray, q_hat4, '-k'),grid on;
%legend('qhat(1)','qhat(2)','qhat(3)', 'qhat(4)');title('attitude quaternion components versus time step');
%pause; close;

%plot(steparray,errarray,'-r'),grid on;
%legend('MSE');title('Magnitude of squared error at each time step');
%pause; close;

%end of function
```

A.9 Mean_Square_Error.m

Mean_square_error.m is a user defined MATLAB function to determine the mean square error (MSE) between two vectors. It is used extensively throughout these algorithms to determine MSE as a measure of method performance.

```
%Mean_square_error - user defined function to calculate mean square error between two
%vectors.
%
%function [mean_square_error]=Mean_square_error(vec1,vec2)
%
%mean_square_error = mse between two vectors of interest
%vec1, vec2 = two vectors of interest
%
%Written by Mark Charlton.  Last updated 21 October 03.

function [mean_square_error]=Mean_square_error(vec1,vec2)

sum=0;
for i=1:length(vec1)
   diffsquared=(vec1(i)-vec2(i)).^2;
   sum=sum+diffsquared;
end
mean_square_error=sum/length(vec1);
```

A.10 Qframerot6.m

Qframerot6.m is a user defined MATLAB function to take a six-element input vector through a frame rotation and output the rotated vector.

```
%Qframerot6 - user defined function to accomplish a frame rotation through the
%multiplication of quaternions.  Yields original 6D vector, expressed in the
%coordinates of the rotated frame, when provided with unitary rotation quaternion.
%
%function [vrot]=qframerot6(q_rot, vinit)
%
%q_rot = rotation quaternion of unit magnitude
%
%vrot = 6D vector expressed in the rotated frame
%vinit = 6D vector expressed in original frame
%
```

```
%Written by Mark Charlton.  Last updated 28 October 03.

function [vrot] = qframerot6(q_rot,vinit)

%Calculate rotation generated by q_rot

%alpha=2*acos(q_rot(4));
%clc
%'Transpose of vector in the original frame is: ', vinit'

%'Rotation to be accomplished in degrees: ',alpha*180/pi
%' '
%axis=[q_rot(1) q_rot(2) q_rot(3)]';
%axis=axis/sqrt(axis'*axis);

%'The transpose of the unit vector representing the Euler axis of rotation: ',axis'

%Quaternion direction cosine matrix describing transformation from
%rotated frame to reference frame [41, pg 7 - 9]

%R_hat(1,1)=q_rot(1)^2-q_rot(2)^2-q_rot(3)^2+q_rot(4)^2;
%R_hat(1,2)=2*(q_rot(1)*q_rot(2)-q_rot(3)*q_rot(4));
%R_hat(1,3)=2*(q_rot(1)*q_rot(3)+q_rot(2)*q_rot(4));
%R_hat(2,1)=2*(q_rot(1)*q_rot(2)+q_rot(3)*q_rot(4));
%R_hat(2,2)=-q_rot(1)^2+q_rot(2)^2-q_rot(3)^2+q_rot(4)^2;
%R_hat(2,3)=2*(-q_rot(1)*q_rot(4)+q_rot(2)*q_rot(3));
%R_hat(3,1)=2*(q_rot(1)*q_rot(3)-q_rot(2)*q_rot(4));
%R_hat(3,2)=2*(q_rot(1)*q_rot(4)+q_rot(2)*q_rot(3));
%R_hat(3,3)=-q_rot(1)^2-q_rot(2)^2+q_rot(3)^2+q_rot(4)^2;

%Transpose gives frame rotation versus vector rotation (describes
%transformation from reference frame to rotated frame...ehat=R*Ehat

%R_hat=R_hat';

%M_hat=[R_hat zeros(3,3);zeros(3,3) R_hat];

%Rotation using equivalent matrix

%vrot_matrix=M_hat*vinit;

%'The rotated vector, expressed in the original frame, using an equivalent matrix is:  ', vrot_matrix
%' '
```

```
%Define conjugate of q_rot

q_rot_conj=[-q_rot(1) -q_rot(2) -q_rot(3) q_rot(4)]';

%Augment vector with zeros to form two pure quaternions from two 3D vectors in vinit

VINIT=[vinit(1:3);0;vinit(4:6);0];

%Rotation using quaternion multiplication

VINITprimeA=qmult(q_rot_conj, VINIT(1:4));     %rotate first half of vector
VROTA=qmult(VINITprimeA, q_rot);

VINITprimeB=qmult(q_rot_conj, VINIT(5:8));     %rotate second half of vector
VROTB=qmult(VINITprimeB, q_rot);

%Pull 3D vector from pure quaternion

vrot=[VROTA(1) VROTA(2) VROTA(3) VROTB(1) VROTB(2) VROTB(3)]';

%'The original vector, expressed in the coordinates of the rotated frame, is: '

%end of function
```

A.11 Qvecrot3.m

This user defined function takes an initial three-element vector and rotates the vector according to an input rotation quaternion. The new vector is represented in the coordinates of the original reference frame.

```
%qvecrot3 - user defined function to accomplish a 3D vector rotation through the
%multiplication of quaternions. Yields rotated 3D vector, expressed in the
%coordinates of the original frame, when provided with unitary rotation quaternion.
%
%function [vrot]=qvecrot3(q_rot, vinit)
%
%q_rot = rotation quaternion of unit magnitude

%vrot = rotated 3D vector expressed in original frame
%vinit = 3D vector to be rotated, expressed in original frame

%Written by Mark Charlton. Last updated 22 October 03.

function [vrot] = qvecrot3(q_rot,vinit)
```

```
%Calculate rotation generated by q_rot

alpha=2*acos(q_rot(4));
clc
%'Vector to be rotated is: ', vinit

%'Rotation to be accomplished in degrees: ',alpha*180/pi
%' '
axis=[q_rot(1) q_rot(2) q_rot(3)]';
axis=axis/sqrt(axis'*axis);

%'The unit vector representing the Euler axis of rotation: ',axis

%Quaternion direction cosine matrix describing transformation from
%rotated frame to reference frame [[40], pg 7 - 9]

R_hat(1,1)=q_rot(1)^2-q_rot(2)^2-q_rot(3)^2+q_rot(4)^2;
R_hat(1,2)=2*(q_rot(1)*q_rot(2)-q_rot(3)*q_rot(4));
R_hat(1,3)=2*(q_rot(1)*q_rot(3)+q_rot(2)*q_rot(4));
R_hat(2,1)=2*(q_rot(1)*q_rot(2)+q_rot(3)*q_rot(4));
R_hat(2,2)=-q_rot(1)^2+q_rot(2)^2-q_rot(3)^2+q_rot(4)^2;
R_hat(2,3)=2*(-q_rot(1)*q_rot(4)+q_rot(2)*q_rot(3));
R_hat(3,1)=2*(q_rot(1)*q_rot(3)-q_rot(2)*q_rot(4));
R_hat(3,2)=2*(q_rot(1)*q_rot(4)+q_rot(2)*q_rot(3));
R_hat(3,3)=-q_rot(1)^2-q_rot(2)^2+q_rot(3)^2+q_rot(4)^2;

%Rotation using equivalent matrix

vrot_matrix=R_hat*vinit;

%'The rotated vector, expressed in the original frame, using an equivalent matrix is:  ', vrot_matrix
%' '

%Define conjugate of q_rot

q_rot_conj=[-q_rot(1) -q_rot(2) -q_rot(3) q_rot(4)]';

%Form pure quaternion from 3D vector vinit

VINIT=[vinit;0];

%Vector rotation using quaternion multiplication
```

```
VINITprime=qmult(q_rot, VINIT);
VROT=qmult(VINITprime, q_rot_conj);

%Pull 3D vector from pure quaternion

vrot=[VROT(1) VROT(2) VROT(3)]';

%'The rotated vector, expressed in the original frame: '

%end of function
```

A.12 Orientation3.m

This user-defined function determines the inertial orientation of the body-fixed frame after rotation by both a "true" attitude quaternion and by an estimated quaternion. It then calculates the angle between each of the axes rotated by each method to form "error angles." This function is called by norates_ekf.m, norates_ukf.m, rates_ekf.m, or by rates_ukf.m if the option is selected to calculate error angles.

```
%Orientation3 - user defined function that determines the inertial orientation
%of the body frame using both an estimated rotation quaternion and a "true"
%quaternion. Body frame vectors are expressed in inertial cartesian
%coordinates. Also calculates "error angles" between body frame unit
%vectors rotated using estimated quaternion and those rotated using "true"
%quaternion.
%
%function [ovec1,ovec2,ovec3,ovec1hat,ovec2hat,ovec3hat,ang1,ang2,ang3]=orientation3(qtrue, qhat)
%
%ovec1 - body frame "x" axis unit vector rotated using "true" quaternion and expressed in inertial coordinates
%ovec2 - body frame "y" axis unit vector rotated using "true" quaternion and expressed in inertial coordinates
%ovec3 - body frame "z" axis unit vector rotated using "true" quaternion and expressed in inertial coordinates
%ovechat1 - body frame "x" axis unit vector rotated using estimated quaternion and expressed in inertial coordinates
%ovechat2 - body frame "y" axis unit vector rotated using estimated quaternion and expressed in inertial coordinates
%ovechat3 - body frame "z" axis unit vector rotated using estimated quaternion and expressed in inertial coordinates
%ang1 - angle between "x" body axis unit vector rotated using "true" and estiamated quaternion
%ang2 - angle between "y" body axis unit vector rotated using "true" and estiamated quaternion
%ang3 - angle between "z" body axis unit vector rotated using "true" and estiamated quaternion
%qtrue - array with true quaternions as columns
%qhat - array with estimated quaternions as columns
%
%Written by Mark Charlton. Last updated 31 October 2003.

function [ovec1,ovec2,ovec3,ovec1hat,ovec2hat,ovec3hat,ang1,ang2,ang3]=orientation3(qtrue, qhat)
```

```
format compact
omat=[ ];
omathat=[ ];

body1=[1 0 0]';
body2=[0 1 0]';
body3=[0 0 1]';

for j=1:length(qhat)
   ovec1(1:3,j)=qvecrot3(qtrue(1:4,j),body1);
   ovec2(1:3,j)=qvecrot3(qtrue(1:4,j),body2);
   ovec3(1:3,j)=qvecrot3(qtrue(1:4,j),body3);
   ovec1hat(1:3,j)=qvecrot3(qhat(1:4,j),body1);
   ovec2hat(1:3,j)=qvecrot3(qhat(1:4,j),body2);
   ovec3hat(1:3,j)=qvecrot3(qhat(1:4,j),body3);
end

%*************** Plotting and output statements *****************

PLOTS=input('Enter "1" to display plots. Enter "0" for no plots. Default is no plots:  ');
if isempty(PLOTS)
  PLOTS=0;
end

if PLOTS==1
   %normalize estimates of orientation vectors for ease of plot interpretation

%   for k=1:length(ovec1hat)
%      ovec1hat(1:3,k)=ovec1hat(1:3,k)./norm(ovec1hat(1:3,k));
%      ovec2hat(1:3,k)=ovec2hat(1:3,k)./norm(ovec2hat(1:3,k));
%      ovec3hat(1:3,k)=ovec3hat(1:3,k)./norm(ovec3hat(1:3,k));
%   end



   plot3(ovec1(1,:),ovec1(2,:),ovec1(3,:),'r');hold on;grid on;plot3(ovec1hat(1,:),ovec1hat(2,:),ovec1hat(3,:),'r+')
   plot3(ovec2(1,:),ovec2(2,:),ovec2(3,:),'g');hold on;grid on;plot3(ovec2hat(1,:),ovec2hat(2,:),ovec2hat(3,:),'g+')
   plot3(ovec3(1,:),ovec3(2,:),ovec3(3,:),'o');hold on;grid on;plot3(ovec3hat(1,:),ovec3hat(2,:),ovec3hat(3,:),'b')
   title('3D plot of "true" and "estimated" body axes in inertial frame');
   xlabel('inertial "X" axis');ylabel('inertial "Y" axis');zlabel('inertial "Z" axis');
   legend('real body x axis','est. body x axis','real body y axis','est. body y axis','real body "z" axis','est body "z" axis');

   'press "enter" to continue...'
```

```
    pause;close;clc;
end

%calculate angles between body axes rotated using "true" quaternion and
%using "estimated" quaternion at each time step.

for j=1:length(ovec1)
    ang1(j)=acos(dot(ovec1(1:3,j),ovec1hat(1:3,j))/(norm(ovec1(1:3,j))*norm(ovec1hat(1:3,j))))*180/pi;
    ang2(j)=acos(dot(ovec2(1:3,j),ovec2hat(1:3,j))/(norm(ovec2(1:3,j))*norm(ovec2hat(1:3,j))))*180/pi;
    ang3(j)=acos(dot(ovec3(1:3,j),ovec3hat(1:3,j))/(norm(ovec3(1:3,j))*norm(ovec3hat(1:3,j))))*180/pi;
end

'mean ang1',mean(ang1)
'std ang1',std(ang1)
'max ang1',max(ang1)
'mean ang2',mean(ang2)
'std ang2',std(ang2)
'max ang2',max(ang2)
'mean ang3',mean(ang3)
'std ang3',std(ang3)
'max ang3',max(ang3)


'processing complete'
pause;close;clc

%end of function
```

A.13 Norates_ekf.m

Norates_ekf.m is the MATLAB script that implements the extended Kalman filter (EKF) designed in Chapter 9. This implementation is for the case where rotational rates are not directly measured, but are derived from quaternion rates. Norates_ekf.m is called by attitude_filter.m if the appropriate option is selected by the user.

```
%NORATES_EKF - Discrete Extended Kalman Filter Implementation
%Uses data generated by attitude_truth.m to estimate 3-D body rates and
%the attitude quaternion of a sounding rocket given noisy measurements
%from two attitude sensors. Uses a Gauss-Newton optimization and an
%Extended Kalman Filter (EKF) to generate a best estimate.
%
%Written by Mark Charlton. Last modified 28 October 03.
```

```
%Object subject to rotational motion -- non-linear Equation of Motion

%***** Initialize parameters and matrices *****

clc

ics=[ ];
sigmas=[ ];
error=[ ];                                                        %initialize error multiplier
sensorerror=[ ];                                                  %initialize error multiplier
ArrayT=[ ];                                                       %initialize Arrays
q1hat=[ ];
q2hat=[ ];
q3hat=[ ];
q4hat=[ ];
omega1hat=[ ];
omega2hat=[ ];
omega3hat=[ ];
ArraySP11=[ ];
ArraySP11P=[ ];
ArraySP22=[ ];
ArraySP22P=[ ];
ArraySP33=[ ];
ArraySP33P=[ ];
ArraySP44=[ ];
ArraySP44P=[ ];
ArraySP55=[ ];
ArraySP55P=[ ];
ArraySP66=[ ];
ArraySP66P=[ ];
ArraySP77=[ ];
ArraySP77P=[ ];
H1=.01;                                     %assign integration step-size for Euler integration
statesize=7;                                %system order
PHIS=0;                                     %process noise (measure of uncertainty in dynamic model)
PHI=zeros(statesize,statesize);             %initialize state transition matrix, PHI
Q=zeros(statesize,statesize);               %initialize process noise matrix
P=zeros(statesize,statesize);               %initialize covariance matrix
HMAT=zeros(statesize,statesize);            %initialize linearized measurement matrix

%***** Assign initial estimates for omega1hat, omega2hat, omega3hat, q1hat, q2hat, q3hat, q4hat *****

'Do you wish to enter initial estimates of the rotational body rates and the quaternion components'
```

```
'or let the program use the actual initial conditions with a user defined percent error to'
'make an initial estimate (default is to let the program do it)?'
''
'If you choose manual entry, an uncertainty in the estimate is also required. The default is "auto".'
''
ics=input('Enter "1" for manual entry and "0" for automatic:  ');
if isempty(ics)
  ics=0;
end

if ics == 0

%***** base initialization on actual intial conditions with user defined standard error *****
clc
error=input('Enter percentage error (+/-) in initial conditions (i.e., -5.2% = "-5.2") Default is 5%:  ');
if isempty(error)
  error=5;
end

'percent error applied to the initial conditions is: ',error

error=error/100;

omega1hat(1)=omega1(1)+omega1(1)*error;

omega2hat(1)=omega2(1)+omega2(1)*error;
omega3hat(1)=omega3(1)+omega3(1)*error;
q1hat(1)=q1(1)+q1(1)*error;
q2hat(1)=q2(1)+q2(1)*error;
q3hat(1)=q3(1)+q3(1)*error;
q4hat(1)=q4(1)+q4(1)*error;

'apriori estimate of omega1 (rad/s): ',omega1hat(1)                 %apriori estimate of omega1
'apriori estimate of omega2 (rad/s): ',omega2hat(1)                 %apriori estimate of omega2
'apriori estimate of omega3 (rad/s): ',omega3hat(1)                 %apriori estimate of omega3
'apriori estimate of q1: ',q1hat(1)                                 %apriori estimate of q1
'apriori estimate of q2: ',q2hat(1)                                 %apriori estimate of q2
'apriori estimate of q3: ',q3hat(1)                                 %apriori estimate of q3
'apriori estimate of q4: ',q4hat(1)                                 %apriori estimate of q4

%***** Non-zero entries of initial covariance matrix for auto initial cond *****

P(1,1)=(omega1(1)*error+eps)^2;                                       %set at square of uncertainty in omega1
P(2,2)=(omega2(1)*error+eps)^2;                                       %set at square of uncertainty in omega2
```

```
P(3,3)=(omega3(1)*error+eps)^2;                                          %set at square of uncertainty in omega3
P(4,4)=(q1(1)*error+eps)^2;                                              %set at square of uncertainty in q1
P(5,5)=(q2(1)*error+eps)^2;                                              %set at square of uncertainty in q2
P(6,6)=(q3(1)*error+eps)^2;                                              %set at square of uncertainty in q3
P(7,7)=(q4(1)*error+eps)^2;                                              %set at square of uncertainty in q4

else

%...or can manually enter whatever intialization values you wish...

omega1hat(1)=input('Please enter initial omega1 in rpm:  ');             %apriori estimate of omega1
omega2hat(1)=input('Please enter initial omega2 in rpm:  ');             %apriori estimate of omega2
omega3hat(1)=input('Please enter initial omega3 in rpm:  ');             %apriori estimate of omega3
q1hat(1)=input('Please enter initial q1:  ');                            %apriori estimate of q1
q2hat(1)=input('Please enter initial q2:  ');                            %apriori estimate of q2
q3hat(1)=input('Please enter initial q3:  ');                            %apriori estimate of q3
q4hat(1)=input('Please enter initial q4:  ');                            %apriori estimate of q4

omega1hat(1)=omega1hat(1)*2*pi/60;
omega2hat(1)=omega2hat(1)*2*pi/60;
omega3hat(1)=omega3hat(1)*2*pi/60;                                        %convert manual omega entries to rad/s

'For manual entry of initial conditions, estimated uncertainty in the initial conditions'
'is necessary to intialize the covariance matrix...'
''

domega1hat=input('Please enter estimated uncertainty in initial omega1 (rad/s):  ');   %uncertainty in initial estimate of omega1
domega2hat=input('Please enter estimated uncertainty in initial omega2 (rad/s):  ');   %uncertainty in initial estimate of omega2
domega3hat=input('Please enter estimated uncertainty in initial omega3 (rad/s):  ');   %uncertainty in initial estimate of omega3
dq1hat=input('Please enter estimated uncertainty in initial q1:  ');                   %uncertainty in initial estimate of q1
dq2hat=input('Please enter estimated uncertainty in initial q2:  ');                   %uncertainty in initial estimate of q2
dq3hat=input('Please enter estimated uncertainty in initial q3:  ');                   %uncertainty in initial estimate of q3
dq4hat=input('Please enter estimated uncertainty in initial q4:  ');                   %uncertainty in initial estimate of q4

%***** Non-zero entries of initial covariance matrix for manual initial cond *****

P(1,1)=(domega1hat+eps)^2;                      %set at square of uncertainty in initial estimate of omega1
P(2,2)=(domega2hat+eps)^2;                      %set at square of uncertainty in initial estimate of omega2
P(3,3)=(domega3hat+eps)^2;                      %set at square of uncertainty in initial estimate of omega3
P(4,4)=(dq1hat+eps)^2;                          %set at square of uncertainty in initial estimate of q3
P(5,5)=(dq2hat+eps)^2;                          %set at square of uncertainty in initial estimate of q3
P(6,6)=(dq3hat+eps)^2;                          %set at square of uncertainty in initial estimate of q3
P(7,7)=(dq4hat+eps)^2;                          %set at square of uncertainty in initial estimate of q4

end
```

```
pause;clc

%insure initial quaternion estimate is a unit quaternion

mag=sqrt(q1hat(1)^2+q2hat(1)^2+q3hat(1)^2+q4hat(1)^2);
q1hat(1)=q1hat(1)/mag;
q2hat(1)=q2hat(1)/mag;
q3hat(1)=q3hat(1)/mag;
q4hat(1)=q4hat(1)/mag;

%***** Enter measurement noise values for use in calculating Measurement Noise Matrix, R *****

'Do you wish to enter sensor measurement sigmas manually or let the program use the actual'
'sensor measurement sigmas with a user defined percent error to make an initial estimate'
'(default is to let the program do it)?'
' '
sigmas=input('Enter "1" for manual entry and "0" for automatic:  ');
if isempty(sigmas)
  sigmas=0;
end
clc
if sigmas == 0

%***** base sensor noise matrix on actual sigmas plus user defined error *****

sensorerror=input('Enter percentage error (+/-) in sensor measurement sigmas (i.e., -5.2% = "-5.2") Default is 0%:  ');
if isempty(sensorerror)
  sensorerror=0;
end

'percent error applied to the actual sensor measurement sigmas is: ',sensorerror

sensorerror=sensorerror/100;

SIGOMEGA1=stdomega1;                        %right now...calculated in attitude_filter from std of actual error
SIGOMEGA2=stdomega2;                        %right now...calculated in attitude_filter from std of actual error
SIGOMEGA3=stdomega3;                        %right now...calculated in attitude_filter from std of actual error
estSIGOMEGA1=SIGOMEGA1+SIGOMEGA1*sensorerror;
estSIGOMEGA2=SIGOMEGA2+SIGOMEGA2*sensorerror;
estSIGOMEGA3=SIGOMEGA3+SIGOMEGA3*sensorerror;
estSIGSUNX=SIGSUNX+SIGSUNX*sensorerror;
estSIGSUNY=SIGSUNY+SIGSUNY*sensorerror;
estSIGSUNZ=SIGSUNZ+SIGSUNZ*sensorerror;
estSIGMAGX=SIGMAGX+SIGMAGX*sensorerror;
```

```
estSIGMAGY=SIGMAGY+SIGMAGY*sensorerror;
estSIGMAGZ=SIGMAGZ+SIGMAGZ*sensorerror;

'apriori estimate of omega1 "measurement" sigma (rad/s): ',estSIGOMEGA1              %apriori estimate of SIGOMEGA1
'apriori estimate of omega2 "measurement" sigma (rad/s): ',estSIGOMEGA2              %apriori estimate of SIGOMEGA2
'apriori estimate of omega3 "measurement" sigma (rad/s): ',estSIGOMEGA3              %apriori estimate of SIGOMEGA3
'apriori estimate of sun sensor x axis measurement sigma (deg): ',estSIGSUNX*180/pi    %apriori estimate of SIGSUNX
'apriori estimate of sun sensor y axis measurement sigma (deg): ',estSIGSUNY*180/pi    %apriori estimate of SIGSUNY
'apriori estimate of sun sensor z axis measurement sigma (deg): ',estSIGSUNZ*180/pi    %apriori estimate of SIGSUNZ
'apriori estimate of mag field sensor x axis measurement sigma (deg): ',estSIGMAGX*180/pi   %apriori estimate of SIGMAGX
'apriori estimate of mag field sensor y axis measurement sigma (deg): ',estSIGMAGY*180/pi   %apriori estimate of SIGMAGY
'apriori estimate of mag field sensor z axis measurement sigma (deg): ',estSIGMAGZ*180/pi   %apriori estimate of SIGMAGZ

else

%...or can manually enter whatever intialization values you wish...

estSIGOMEGA1=input('Please enter sigma for the omega1 derived "measurement" in rpm:  ');  %apriori estimate of SIGOMEGA1
estSIGOMEGA2=input('Please enter sigma for the omega2 derived "measurement" in rpm:  ');  %apriori estimate of SIGOMEGA2
estSIGOMEGA3=input('Please enter sigma for the omega3 derived "measurement" in rpm:  ');  %apriori estimate of SIGOMEGA3

estSIGOMEGA1=estSIGOMEGA1*2*pi/60;                                                   %convert estSIGOMEGA1 to rad/s
estSIGOMEGA2=estSIGOMEGA2*2*pi/60;                                                   %convert estSIGOMEGA2 to rad/s
estSIGOMEGA3=estSIGOMEGA3*2*pi/60;                                                   %convert estSIGOMEGA3 to rad/s

estSIGSUNX=input('Please enter sigma for the sun sensor x axis measurement in deg:  '); %apriori estimate of SIGSUNX
estSIGSUNY=input('Please enter sigma for the sun sensor y axis measurement in deg:  '); %apriori estimate of SIGSUNY
estSIGSUNZ=input('Please enter sigma for the sun sensor z axis measurement in deg:  '); %apriori estimate of SIGSUNZ
estSIGMAGX=input('Please enter sigma for the mag field sensor x axis measurement in deg:  ');%apriori estimate of SIGMAGX
estSIGMAGY=input('Please enter sigma for the mag field sensor y axis measurement in deg:  ');%apriori estimate of SIGMAGY
estSIGMAGZ=input('Please enter sigma for the mag field sensor z axis measurement in deg:  ');%apriori estimate of SIGMAGZ

estSIGSUNX=estSIGSUNX*pi/180;                                                        %convert from degrees to radians
estSIGSUNY=estSIGSUNY*pi/180;                                                        %convert from degrees to radians
estSIGSUNZ=estSIGSUNZ*pi/180;                                                        %convert from degrees to radians
estSIGMAGX=estSIGMAGX*pi/180;                                                        %convert from degrees to radians
estSIGMAGY=estSIGMAGY*pi/180;                                                        %convert from degrees to radians
estSIGMAGZ=estSIGMAGZ*pi/180;                                                        %convert from degrees to radians

end

'Please press "return" to continue...'
pause;clc;close;
```

```
%***** Non-zero submatrices of Measurement Noise Matrix, R *****

ROMEGA=diag([estSIGOMEGA1^2+eps  estSIGOMEGA2^2+eps   estSIGOMEGA3^2+eps]);
RSENSOR=diag([estSIGSUNX^2+eps  estSIGSUNY^2+eps   estSIGSUNZ^2+eps   estSIGMAGX^2+eps   estSIGMAGY^2+eps
estSIGMAGZ^2+eps]);

%***** user input of process noise to reflect uncertainty in dynamic model, PHIS *****
''
confidence=input('Enter process noise to reflect uncertainty in model (default is 3.53):  ');

if isempty(confidence)
  confidence=3.53;
end

PHIS=confidence;          %(100-confidence)/100;                          %[17] pg. 317

'Process noise applied is: ',PHIS

'press "return" to continue...'
pause;clc;close;

%***** loop to calculate estimated states at each time step *****

count=1;
                                                                          %initialize counter
t=0.;
                                                                          %initialize time
ArrayT(1)=t;

%***** time constants and noise variances for noise driving motion *****
%***** right now, the time constants are hard-coded *****
'enter time constants associated with motion...'
''

tauomega1 = input('enter tau omega1 (default=29)    ');
if isempty(tauomega1)
   tauomega1=29
end
tauomega2 = input('enter tau omega2 (default = 29)    ');
if isempty(tauomega2)
   tauomega2=29
end
tauomega3 = input('enter tau omega3 (default=1e7)    ');
if isempty(tauomega3)
```

```
  tauomega3=1e7
end

while count<=length(omega1_q_min)-1


  %***** Assign variables to more managable variable names *****

  x1=omega1hat(count);
  x2=omega2hat(count);
  x3=omega3hat(count);
  x4=q1hat(count);
  x5=q2hat(count);
  x6=q3hat(count);
  x7=q4hat(count);

  den=sqrt(x4^2+x5^2+x6^2+x7^2);          %convenient compilation of terms to be used later
  den3=den^3;                              %convenient compilation of terms to be used later
  f4=(x1*x7-x2*x6+x3*x5);                  %convenient compilation of terms to be used later
  f5=(x1*x6+x2*x7-x3*x4);                  %convenient compilation of terms to be used later
  f6=(-x1*x5+x2*x4+x3*x7);                 %convenient compilation of terms to be used later
  f7=(-x1*x4-x2*x5-x3*x6);                  %convenient compilation of terms to be used later

  %***** Calculate elements of linearized State Transition Matrix, F *****

  F(1,1)=-1/tauomega1;          %dx1x1
  F(1,2)=0;                     %dx1x2
  F(1,3)=0;                     %dx1x3
  F(1,4)=0;                     %dx1x4
  F(1,5)=0;                     %dx1x5
  F(1,6)=0;                     %dx1x6
  F(1,7)=0;                     %dx1x7
  F(2,1)=0;                     %dx2x1
  F(2,2)=-1/tauomega2;          %dx2x2
  F(2,3)=0;                     %dx2x3
  F(2,4)=0;                     %dx2x4
  F(2,5)=0;                     %dx2x5
  F(2,6)=0;                     %dx2x6
  F(2,7)=0;                     %dx2x7
  F(3,1)=0;                     %dx3x1
  F(3,2)=0;                     %dx3x2
  F(3,3)=-1/tauomega3;          %dx3x3
  F(3,4)=0;                     %dx3x4
  F(3,5)=0;                     %dx3x5
```

```
F(3,6)=0;                       %dx3x6
F(3,7)=0;                       %dx3x7
F(4,1)=.5*x7/den;               %dx4x1
F(4,2)=.5*(-x6)/den;                    %dx4x2
F(4,3)=.5*x5/den;                       %dx4x3
F(4,4)=.5*f4*(-x4)/den3;                %dx4x4
F(4,5)=.5*(x3/den+f4*(-x5)/den3);       %dx4x5
F(4,6)=.5*(-x2/den-f4*x6/den3);         %dx4x6
F(4,7)=.5*(x1/den+f4*(-x7)/den3);       %dx4x7
F(5,1)=.5*x6/den;                       %dx5x1
F(5,2)=.5*x7/den;                       %dx5x2
F(5,3)=.5*(-x4)/den;                    %dx5x3
F(5,4)=.5*(-x3/den+f5*(-x4)/den3);      %dx5x4
F(5,5)=.5*f5*(-x5)/den3;                %dx5x5
F(5,6)=.5*(x1/den+f5*(-x6)/den3);       %dx5x6
F(5,7)=.5*(x2/den+f5*(-x7)/den3);       %dx5x7
F(6,1)=.5*(-x5)/den;                    %dx6x1
F(6,2)=.5*x4/den;                       %dx6x2
F(6,3)=.5*x7/den;                       %dx6x3
F(6,4)=.5*(x2/den+f6*(-x4)/den3);       %dx6x4
F(6,5)=.5*(-x1/den+f6*(-x5)/den3);      %dx6x5
F(6,6)=.5*f6*(-x6)/den3;                %dx6x6
F(6,7)=.5*(x3/den+f6*(-x7)/den3);       %dx6x7
F(7,1)=.5*(-x4)/den;                    %dx7x1
F(7,2)=.5*(-x5)/den;                    %dx7x2
F(7,3)=.5*(-x6)/den;                    %dx7x3
F(7,4)=.5*(-x1/den+f7*(-x4)/den3);      %dx7x4
F(7,5)=.5*(-x2/den+f7*(-x5)/den3);      %dx7x5
F(7,6)=.5*(-x3/den+f7*(-x6)/den3);      %dx7x6
F(7,7)=.5*f7*(-x7)/den3;                %dx7x7

%*** Use first two terms of infinite Taylor Series expansion to form non-zero elements of Discrete Fundamental Matrix, PHI ***

PHI=eye(7)+F*t;

%***** Calculate non-zero elements of Discrete Process Noise Covariance Matrix (old Q)*****

num1=(.5*TS^2-TS^3/(3*tauomega1));      %convenient compilation of terms to be used later
num2=(.5*TS^2-TS^3/(3*tauomega2));      %convenient compilation of terms to be used later
num3=(.5*TS^2-TS^3/(3*tauomega3));      %convenient compilation of terms to be used later

Q(1,1)=estSIGOMEGA1^2*(TS-TS^2/tauomega1+TS^3/(3*tauomega1^2));
Q(1,4)=estSIGOMEGA1^2*F(4,1)*num1;
Q(1,5)=estSIGOMEGA1^2*F(1,5)*num1;
```

```
Q(1,6)=estSIGOMEGA1^2*F(1,6)*num1;
Q(1,7)=estSIGOMEGA1^2*F(1,7)*num1;
Q(2,2)=estSIGOMEGA2^2*(TS-TS^2/tauomega2*TS^3/(3*tauomega2));
Q(2,4)=estSIGOMEGA2^2*F(4,2)*num2;
Q(2,5)=estSIGOMEGA2^2*F(5,2)*num2;
Q(2,6)=estSIGOMEGA2^2*F(6,2)*num2;
Q(2,7)=estSIGOMEGA2^2*F(7,2)*num2;
Q(3,3)=estSIGOMEGA3^2*(TS-TS^2/tauomega3*TS^3/(3*tauomega3));
Q(3,4)=estSIGOMEGA3^2*F(4,3)*num3;
Q(3,5)=estSIGOMEGA3^2*F(5,3)*num3;
Q(3,6)=estSIGOMEGA3^2*F(6,3)*num3;
Q(3,7)=estSIGOMEGA3^2*F(7,3)*num3;
Q(4,1)=Q(1,4);
Q(4,2)=Q(2,4);
Q(4,3)=Q(3,4);
Q(4,4)=estSIGOMEGA1^2*(F(4,1)^2+estSIGOMEGA2^2*F(4,2)^2+estSIGOMEGA3^2*F(4,3)^2)*TS^3/3;
Q(5,1)=Q(1,5);
Q(5,2)=Q(2,5);
Q(5,3)=Q(3,5);
Q(5,4)=Q(4,5);
Q(5,5)=estSIGOMEGA1^2*(F(5,1)^2+estSIGOMEGA2^2*F(5,2)^2+estSIGOMEGA3^2*F(5,3)^2)*TS^3/3;
Q(5,6)=(estSIGOMEGA1^2*F(5,1)*F(6,1)+estSIGOMEGA2^2*F(5,2)*F(6,2)+estSIGOMEGA3^2*F(5,3)*F(6,3))*TS^3/3;
Q(5,7)=(estSIGOMEGA1^2*F(5,1)*F(7,1)+estSIGOMEGA2^2*F(5,2)*F(7,2)+estSIGOMEGA3^2*F(5,3)*F(7,3))*TS^3/3;
Q(6,1)=Q(1,6);
Q(6,2)=Q(2,6);
Q(6,3)=Q(3,6);
Q(6,4)=Q(4,6);
Q(6,5)=Q(5,6);
Q(6,6)=estSIGOMEGA1^2*(F(6,1)^2+estSIGOMEGA2^2*F(6,2)^2+estSIGOMEGA3^2*F(6,3)^2)*TS^3/3;
Q(6,7)=(estSIGOMEGA1^2*F(6,1)*F(7,1)+estSIGOMEGA2^2*F(6,2)*F(7,2)+estSIGOMEGA3^2*F(6,3)*F(7,3))*TS^3/3;
Q(7,1)=Q(1,7);
Q(7,2)=Q(2,7);
Q(7,3)=Q(3,7);
Q(7,4)=Q(4,7);
Q(7,5)=Q(5,7);
Q(7,6)=Q(6,7);
Q(7,7)=estSIGOMEGA1^2*(F(7,1)^2+estSIGOMEGA2^2*F(7,2)^2+estSIGOMEGA3^2*F(7,3)^2)*TS^3/3;

Q=PHIS*Q;

%Q=diag([estSIGOMEGA1 estSIGOMEGA2 estSIGOMEGA3 0 0 0 0]);   %constant values of Q...zeros for quaternion
                                                             %values since these are not noise driven, directly  [11]


%***** Calculate covariance before update, M *****
```

```
PHIT=PHI';                                          %calculate transpose of PHI;
PHIP=PHI*P;
PHIPPHIT=PHIP*PHIT;

M=PHIPPHIT+Q;                               %calculate covariance before update
                                            %(using old P, old PHI, and old Q)

t=t+TS;                                     %increment time

%***** Propagate estimated states forward using one-step Euler integration *****

S=0;                                        %initialize integration timer

while S<=(TS-.0001)                         %loops from current step to just short of next time step

  x1dot=-1/tauomega1*x1;
  x1=x1+H1*x1dot;

  x2dot=-1/tauomega2*x2;
  x2=x2+H1*x2dot;

  x3dot=-1/tauomega3*x3;
  x3=x3+H1*x3dot;

  x4dot=.5*f4/den;
  x4=x4+H1*x4dot;

  x5dot=.5*f5/den;
  x5=x5+H1*x5dot;

  x6dot=.5*f6/den;
  x6=x6+H1*x6dot;

  x7dot=.5*f7/den;
  x7=x7+H1*x7dot;

  S=S+H1;                                   %increment integration timer by defined integration step size
end                                         %yields each state estimate integrated forward to next time step

%***** assign values from numerical integration to arrays *****

xbar=[x1;x2;x3;x4;x5;x6;x7];
```

```
omega1bar(count+1)=x1;
omega2bar(count+1)=x2;
omega3bar(count+1)=x3;
q1bar(count+1)=x4;
q2bar(count+1)=x5;
q3bar(count+1)=x6;
q4bar(count+1)=x7;

%***** Calculate non-zero elements of the linearized measurement matrix *****

HMAT=eye(7);                                    %in this implementation, measurements are the states

%***** Calculate Measurement Noise Matrix, RMAT *****
%***** Uses A matrix from Gauss_newton to relate 6 x 6 RSENSOR to covariance of quaternions after convergence [11] ******

RMAT=[ROMEGA, zeros(3,4);zeros(4,3), A(:,:,count)*RSENSOR*A(:,:,count)'];

%***** Calculate Kalman gain matrix *****

HT=HMAT';                                       %calculate transpose of HT (next time step)

HM=HMAT*M;
HMHT=HM*HT;
HMHTR=HMHT+RMAT;
HMHTRINV=inv(HMHTR);
MHT=M*HT;

GAIN=MHT*HMHTRINV;                              %Calculate Kalman gain matrix

    KH=GAIN*HMAT;
    IKH=eye(statesize)-KH;

%***** Calculate covariance matrix after update, P *****

P=IKH*M;                                        %Calculate covariance after update (new P)

%***** Calculate residuals: actual new measurement - projected new measurement *****

%***** Since the H matrix is the identity matrix in this implementation, the "zbar", *****
%***** or projected measurement, is simply the projected state (zbar=xbar)! *****

omega1res=(omega1_q_min(count+1)-x1);           %calculate residual from propagated state and measurement
omega2res=(omega2_q_min(count+1)-x2);            %calculate residual from propagated state and measurement
omega3res=(omega3_q_min(count+1)-x3);           %calculate residual from propagated state and measurement
```

```
q1res=q_min(1,count+1)-x4;                         %calculate residual from propagated state and measurement
q2res=q_min(2,count+1)-x5;                         %calculate residual from propagated state and measurement
q3res=q_min(3,count+1)-x6;                         %calculate residual from propagated state and measurement
q4res=q_min(4,count+1)-x7;                         %calculate residual from propagated state and measurement

residual=[omega1res;omega2res;omega3res;q1res;q2res;q3res;q4res];            %form vector of residuals

%***** Calculate updated state estimates using propagated states and Kalman gains

xhat=xbar+GAIN*residual;

%***** Check for loss of data *****

if q_min(1:4,count)==[0 0 0 1]';
  omega1hat(count+1)=omega1hat(count);
  omega2hat(count+1)=omega2hat(count);
  omega3hat(count+1)=omega3hat(count);
  q1hat(count+1)=q1hat(count);
  q2hat(count+1)=q2hat(count);
  q3hat(count+1)=q3hat(count);
  q4hat(count+1)=q4hat(count);
  flag(count)=1;

else

%*************** Data management *****************

  omega1hat(count+1)=xhat(1);
  omega2hat(count+1)=xhat(2);
  omega3hat(count+1)=xhat(3);
  q1hat(count+1)=xhat(4);
  q2hat(count+1)=xhat(5);
  q3hat(count+1)=xhat(6);
  q4hat(count+1)=xhat(7);
  flag(count)=0;
end
ArrayT(count+1)=t;

%*************** Error Analysis ******************

SP11=sqrt(P(1,1));                                 %Calculate theoretical upper error bound on omega1hat
SP11P=-SP11;                                       %Calculate theoretical lower error bound on omega1hat
SP22=sqrt(P(2,2));                                 %Calculate theoretical upper error bound on omega2hat
SP22P=-SP22;                                       5Calculate theoretical lower error bound on omega2hat
```

```
  SP33=sqrt(P(3,3));                        %Calculate theoretical upper error bound on omega3hat
  SP33P=-SP33;                              %Calculate theoretical lower error bound on omega3hat
  SP44=sqrt(P(4,4));                        %Calculate theoretical upper error bound on q1hat
  SP44P=-SP44;                                  %Calculate theoretical lower error bound on q1hat
  SP55=sqrt(P(5,5));                            %Calculate theoretical upper error bound on q2hat
  SP55P=-SP55;                                  %Calculate theoretical lower error bound on q2hat
  SP66=sqrt(P(6,6));                            %Calculate theoretical upper error bound on q3hat
  SP66P=-SP66;                                  %Calculate theoretical lower error bound on q3hat
  SP77=sqrt(P(7,7));                            %Calculate theoretical upper error bound on q4hat
  SP77P=-SP77;                                  %Calculate theoretical lower error bound on q4hat

  ArraySP11(count+1)=SP11;
  ArraySP11P(count+1)=SP11P;
  ArraySP22(count+1)=SP22;
  ArraySP22P(count+1)=SP22P;
  ArraySP33(count+1)=SP33;
  ArraySP33P(count+1)=SP33P;
  ArraySP44(count+1)=SP44;
  ArraySP44P(count+1)=SP44P;
  ArraySP55(count+1)=SP55;
  ArraySP55P(count+1)=SP55P;
  ArraySP66(count+1)=SP66;
  ArraySP66P(count+1)=SP66P;
  ArraySP77(count+1)=SP77;
  ArraySP77P(count+1)=SP77P;

  count=count+1;                                %increment counter

end

%****** calculate mean square errors for "instantaneous" and for EKF ******
%****** mean square error excludes first 5 samples to reduce effects of transients *****

s2=length(omega1_q_min);
s1=5;

omega1_q_min_mse=mean_square_error(omega1_q_min(s1:s2),omega1(s1:s2));
omega2_q_min_mse=mean_square_error(omega2_q_min(s1:s2),omega2(s1:s2));
omega3_q_min_mse=mean_square_error(omega3_q_min(s1:s2),omega3(s1:s2));
q1_q_min_mse=mean_square_error(q_min(1,(s1:s2)),q1(s1:s2));
q2_q_min_mse=mean_square_error(q_min(2,(s1:s2)),q2(s1:s2));
q3_q_min_mse=mean_square_error(q_min(3,(s1:s2)),q3(s1:s2));
q4_q_min_mse=mean_square_error(q_min(4,(s1:s2)),q4(s1:s2));
```

```
omega1_ekf_mse=mean_square_error(omega1hat(s1:s2),omega1(s1:s2));
omega2_ekf_mse=mean_square_error(omega2hat(s1:s2),omega2(s1:s2));
omega3_ekf_mse=mean_square_error(omega3hat(s1:s2),omega3(s1:s2));
q1_ekf_mse=mean_square_error(q1hat(s1:s2),q1(s1:s2));
q2_ekf_mse=mean_square_error(q2hat(s1:s2),q2(s1:s2));
q3_ekf_mse=mean_square_error(q3hat(s1:s2),q3(s1:s2));
q4_ekf_mse=mean_square_error(q4hat(s1:s2),q4(s1:s2));

%*************** Plotting and output statements *****************

plots=input('Enter "1" to display plots. Enter "0" for no plots. Default is no plots:  ');
if isempty(plots)
  plots=0;
end

if plots==1

%****** plots of error vs. true and theoretical error bounds *******

plot(ArrayT,(omega1(1:s2)-omega1hat),ArrayT,ArraySP11,'x',ArrayT,ArraySP11P,'x');
grid on;title('error in EKF estimate of omega1');
legend('error in omega1hat','theoretical bound');
xlabel('time (sec)');ylabel('error in omega1hat (rad/s)');
pause;close;clc;

plot(ArrayT,(omega2(1:s2)-omega2hat),ArrayT,ArraySP22,'x',ArrayT,ArraySP22P,'x');
grid on;title('error in EKF estimate of omega2');
legend('error in omega2hat','theoretical bound');
xlabel('time (sec)');ylabel('error in omega2hat (rad/s)');
pause;close;clc;

plot(ArrayT,(omega3(1:s2)-omega3hat),ArrayT,ArraySP33,'x',ArrayT,ArraySP33P,'x');
grid on;title('error in EKF estimate of omega3');
legend('error in omega3hat','theoretical bound');
xlabel('time (sec)');ylabel('error in omega3hat (rad/s)');
pause;close;clc;

plot(ArrayT,(q1(1:s2)-q1hat),ArrayT,ArraySP44,'x',ArrayT,ArraySP44P,'x');
grid on;title('error in EKF estimate of q1');
legend('error in q1hat','theoretical bound');
xlabel('time (sec)');ylabel('error in q1');
pause;close;clc;

plot(ArrayT,(q2(1:s2)-q2hat),ArrayT,ArraySP55,'x',ArrayT,ArraySP55P,'x');
```

```
grid on;title('error in EKF estimate of q2');
legend('error in q2hat','theoretical bound');
xlabel('time (sec)');ylabel('error in q2');
pause;close;clc;

plot(ArrayT,(q3(1:s2)-q3hat),ArrayT,ArraySP66,'x',ArrayT,ArraySP66P,'x');
grid on;title('error in EKF estimate of q3');
legend('error in q3hat','theoretical bound');
xlabel('time (sec)');ylabel('error in q3');
pause;close;clc;

plot(ArrayT,(q4(1:s2)-q4hat),ArrayT,ArraySP77,'x',ArrayT,ArraySP77P,'x');
grid on;title('error in EKF estimate of q4');
legend('error in q4hat','theoretical bound');
xlabel('time (sec)');ylabel('error in q4');
pause;close;clc;

%****** plots to show true vs. quaternion derived alone vs. EKF estimates *******

plot( ArrayT, omega1_q_min,'gx',ArrayT, omega1hat,'m', ArrayT, omega1(1:s2),'--');grid on;
title('Quaternion derived omega1, EKF estimate of omega1, "True" omega1' );
xlabel('time (sec)');ylabel('omega1 (rad/s)');
legend('omega1 from differencing quaternions','omega1 from EKF','true omega1');
pause;clc;close;

plot( ArrayT, omega2_q_min,'gx',ArrayT, omega2hat,'m', ArrayT, omega2(1:s2),'--');grid on;
title('Quaternion derived omega2, EKF estimate of omega2, "True" omega2' );
xlabel('time (sec)');ylabel('omega2 (rad/s)');
legend('omega2 from differencing quaternions','omega2 from EKF','true omega2');
pause;clc;close;

plot( ArrayT, omega3_q_min,'gx',ArrayT, omega3hat,'m', ArrayT, omega3(1:s2),'--');grid on;
title('Quaternion derived omega3, EKF estimate of omega3, "True" omega3' );
xlabel('time (sec)');ylabel('omega3 (rad/s)');
legend('omega3 from differencing quaternions','omega3 from EKF','true omega3');
pause;clc;close;

end

'Summary of performance of each method...'
' '
'The MSE in omega1 derived from differencing quaternions is (rad/s): ',' ',omega1_q_min_mse
' '
'The MSE in omega1 for the EKF estimates is (rad/s): ',' ',omega1_ekf_mse
```

```
' '
'The MSE in omega2 derived from differencing quaternions is (rad/s): ',' ',omega2_q_min_mse
' '
'The MSE in omega2 for the EKF estimates is (rad/s): ',' ',omega2_ekf_mse
' '
'Please press "return" to continue...'
pause;clc;close;

'The MSE in omega3 derived from differencing quaternions is (rad/s): ',' ',omega3_q_min_mse
' '
'The MSE in omega3 for the EKF estimates is (rad/s): ',' ',omega3_ekf_mse
' '
'The MSE in q1 derived using Gauss-Newton is: ',' ',q1_q_min_mse
' '
'The MSE in q1  for the EKF estimates is: ',' ',q1_ekf_mse
' '
'Please press "return" to continue...'
pause;clc;close;

'Summary of performance of each method (cont.) ...'
' '
'The MSE in q2 derived using Gauss-Newton is: ',' ',q2_q_min_mse
' '
'The MSE in q2 for the EKF estimates is: ',' ',q2_ekf_mse
' '
'press "return" to continue...'
pause;clc;close;

'Summary of performance of each method (cont.) ...'
' '
'The MSE in q3 derived using Gauss-Newton is: ',' ',q3_q_min_mse
' '
'The MSE in q3 for the EKF estimates is: ',' ',q3_ekf_mse
' '
'The MSE in q4 derived using Gauss-Newton is: ',' ',q4_q_min_mse
' '
'The MSE in q4 for the EKF estimates is: ',' ',q4_ekf_mse
' '
'press "return" to continue...'
pause;clc;close;

'Summary of performance of each method (cont.) ...'

%****** rotate each vector using actual and ekf attitude quaternion ******
```

```
%****** check mean-square-error between vectors ******

'****** Evaluate Mean-Square-Error of rotated vectors using Gauss-Newton derived quaternion ******'
'Press "return" to rotate the reference vector at each time step to the'
'representation in the rotated frame using both the actual rotation quaternion'
'and the quaternion derived using the ekf...once rotation is complete,'
'calculate the mean-square-error between the results.'
pause;close;clc;

qekf=[q1hat;q2hat;q3hat;q4hat];                              %matrix representation of ekf quaternions

ekf_mse_error=[ ];
for j=1:count;
   ekfvec(1:6,j)=qframerot6(qekf(1:4,j),vref(1:6,j));         %rotate ref vector using min error quaternion
   Q_vec(1:6,j)=qframerot6(Qvec(1:4,j),vref(1:6,j));          %rotate ref vector using true quaternion
   ekf_mse_error(j)=mean_square_error(ekfvec(1:6,j),Q_vec(1:6,j)); %calculate MSE between rotated vectors
end

%***** determine angle between rotated vectors *****
for k=1:count
   ekfsunangle(k)=acos(dot(ekfvec(1:3,k),Q_vec(1:3,k))/(norm(ekfvec(1:3,k))*norm(Q_vec(1:3,k))));
   ekfmagangle(k)=acos(dot(ekfvec(4:6,k),Q_vec(4:6,k))/(norm(ekfvec(4:6,k))*norm(Q_vec(4:6,k))));
   GNsunangle(k)=acos(dot(q_minvec(1:3,k),Q_vec(1:3,k))/(norm(q_minvec(1:3,k))*norm(Q_vec(1:3,k))));
   GNmagangle(k)=acos(dot(q_minvec(4:6,k),Q_vec(4:6,k))/(norm(q_minvec(4:6,k))*norm(Q_vec(4:6,k))));
end
ekfsunangle=ekfsunangle*180/pi;
ekfmagangle=ekfmagangle*180/pi;
GNsunangle=GNsunangle*180/pi;
GNmagangle=GNmagangle*180/pi;

%****** Plot vectors rotated using derived and actual quaternions *****
%plot3(Q_vec(1,1:count),Q_vec(2,1:count),Q_vec(3,1:count),'b');grid on;hold on;
%plot3(ekfvec(1,1:count),ekfvec(2,1:count),ekfvec(3,1:count),'m');
%plot3(Q_vec(4,1:count),Q_vec(5,1:count),Q_vec(6,1:count),'r');grid on;hold on;
%plot3(ekfvec(4,1:count),ekfvec(5,1:count),ekfvec(6,1:count),'g');
%xlabel('X component');ylabel('Y component');zlabel('Z component');
%legend('sun vector, actual q','sun vector, ekf q','mag field vector, actual q', 'mag field vector, ekf q');
%title('3-D plot of reference vectors rotated using actual quaternions and ekf derived quaternions');
%pause;close;clc;

%****** Plot mean-square-error between reference vector rotated using actual quaternion ******
%****** and reference vector rotated using derived quaternion.                    ******

%plot(T(1:count),ekf_mse_error(1:count)),grid on;
```

```
%title('mean-square error between reference vectors rotated using ekf derived and actual quaternions');
%xlabel('time (sec)');ylabel('MSE');legend('MSE')

%'press "return" to continue...'
%pause;close;clc;

'******************** Overall Performance Indicator ***********************'
'****** Excludes first five measurements to reduce effect of transients ******'
' '
'The average of the omega mean-square-errors for the differencing method is: '
GNomegaperf=(omega1_q_min_mse+omega2_q_min_mse+omega3_q_min_mse)/3
' '
'The average of the omega mean-square-errors for the ekf results is: '
ekfomegaperf=(omega1_ekf_mse+omega2_ekf_mse+omega3_ekf_mse)/3
' '
'The average of the quaternion component mean-square-errors for the Gauss-Newton method is: '
GNqperf=(q1_q_min_mse+q2_q_min_mse+q3_q_min_mse+q4_q_min_mse)/4
' '
'The average of the quaternion component mean-square-errors for the ekf results is: '
ekfqperf=(q1_ekf_mse+q2_ekf_mse+q3_ekf_mse+q4_ekf_mse)/4
' '
'press "return" to continue...'
pause;clc;close;

'Summary of performance of each method (cont.) ...'
' '
'The average MSE for the ref vectors rotated using GN quaternions: '
mean_GN_mse_error=mean(GN_mse_error)
' '
'The average MSE for the ref vectors rotated using ekf quaternions: '
mean_ekf_mse_error=mean(ekf_mse_error)
' '
'press "return" to continue...'
pause;clc;close;

'Summary of performance of each method (cont.) ...'
' '
'overall GN performance: sqrt[(mean omega MSE)^2 + (mean rotated vector MSE)^2)]'
overall_GN_perf=sqrt(GNomegaperf^2+mean_GN_mse_error^2)
' '
'overall ekf performance: sqrt[(mean omega MSE)^2 + (mean rotated vector MSE)^2)]'
overall_ekf_perf=sqrt(ekfomegaperf^2+mean_ekf_mse_error^2)
' '
%'*********************************************************************************'
```

```
%'******************************** ekf summary **************************************'
%'**************************************************************************************'
'omega:      ',ekfomegaperf
'quaternion:  ',mean_ekf_mse_error
'overall:    ',overall_ekf_perf
'mean sun angle error:   ',mean(ekfsunangle)
'std sun angle error:   ',std(ekfsunangle)
'max sun angle error:   ',max(ekfsunangle)
'mean mag angle error:   ',mean(ekfmagangle)
'std mag angle error:   ',std(ekfmagangle)
'max mag angle error:   ',max(ekfmagangle)
''
'press "return" to continue...'
pause;close;clc;

angles=input('Enter "1" to calculate attitude error angles. Enter "0" for no angles. Default is no angles:  ');
if isempty(angles)
  angles=0;
end

if angles==1

   %Calculate angles between body frame rotated using estimated quaternion and
   %rotated using "true" quaternion

   [ovec1,ovec2,ovec3,ovec1hat,ovec2hat,ovec3hat,ang1,ang2,ang3]=orientation3(Qvec, qekf);

end

%end of program
```

A.14 Norates_ukf.m

Norates_ukf.m is the MATLAB script that implements the unscented Kalman filter (UKF) designed in Chapter 9. This implementation is for the case where rotational rates are not directly measured, but are derived from quaternion rates. Norates_ukf.m is called by attitude_filter.m if the appropriate option is selected by the user.

```
%NORATES_UKF - Discrete Unscented Kalman Filter Implementation
%Uses data generated by attitude_sim.m to estimate 3-D body rates and
%the attitude quaternion of a sounding rocket given noisy measurements
%from two attitude sensors.  Uses a Gauss-Newton optimization and an
```

```
%Unscented Kalman Filter (UKF) to generate a best estimate.
%
%Written by Mark Charlton.  Last modified 28 October 03.

%Object subject to rotational motion -- non-linear Equation of Motion

%***** Initialize parameters and matrices *****

clc

ics=[ ];
sigmas=[ ];
error=[ ];                                    %initialize error multiplier
sensorerror=[ ];                              %initialize error multiplier
ArrayT=[ ];                                   %initialize Arrays
q1hat=[ ];
q2hat=[ ];
q3hat=[ ];
q4hat=[ ];
omega1hat=[ ];
omega2hat=[ ];
omega3hat=[ ];
ArraySP11=[ ];
ArraySP11P=[ ];
ArraySP22=[ ];
ArraySP22P=[ ];
ArraySP33=[ ];
ArraySP33P=[ ];
ArraySP44=[ ];
ArraySP44P=[ ];
ArraySP55=[ ];
ArraySP55P=[ ];
ArraySP66=[ ];
ArraySP66P=[ ];
ArraySP77=[ ];
ArraySP77P=[ ];
statesize=7;                                  %system order
PHIS=0;                                       %process noise (measure of uncertainty in
dynamic model)
Q=zeros(statesize,statesize);                 %initialize process noise matrix
P=zeros(statesize,statesize);                 %initialize covariance matrix
HMAT=zeros(statesize,statesize);              %initialize linearized measurement matrix

%***** Assign initial estimates for omega1hat, omega2hat, omega3hat, q1hat, q2hat, q3hat, q4hat *****
```

```
'Do you wish to enter initial estimates of the rotational body rates and the quaternion components'
'or let the program use the actual initial conditions with a user defined percent error to'
'make an initial estimate (default is to let the program do it)?'
''
'If you choose manual entry, an uncertainty in the estimate is also required. The default is "auto".'
''
ics=input('Enter "1" for manual entry and "0" for automatic:  ');
if isempty(ics)
  ics=0;
end

if ics == 0

%***** base initialization on actual intial conditions with user defined standard error *****
clc
error=input('Enter percentage error (+/-) in initial conditions (i.e., -5.2% = "-5.2") Default is 5%:  ');
if isempty(error)
  error=5;
end

'percent error applied to the initial conditions is: ',error

error=error/100;

omega1hat(1)=omega1(1)+omega1(1)*error;

omega2hat(1)=omega2(1)+omega2(1)*error;
omega3hat(1)=omega3(1)+omega3(1)*error;
q1hat(1)=q1(1)+q1(1)*error;
q2hat(1)=q2(1)+q2(1)*error;
q3hat(1)=q3(1)+q3(1)*error;
q4hat(1)=q4(1)+q4(1)*error;

'apriori estimate of omega1 (rad/s): ',omega1hat(1)          %apriori estimate of omega1
'apriori estimate of omega2 (rad/s): ',omega2hat(1)          %apriori estimate of omega2
'apriori estimate of omega3 (rad/s): ',omega3hat(1)          %apriori estimate of omega3
'apriori estimate of q1: ',q1hat(1)                          %apriori estimate of q1
'apriori estimate of q2: ',q2hat(1)                          %apriori estimate of q2
'apriori estimate of q3: ',q3hat(1)                          %apriori estimate of q3
'apriori estimate of q4: ',q4hat(1)                          %apriori estimate of q4

%***** Non-zero entries of initial covariance matrix for auto initial cond *****
```

```
P(1,1)=(omega1(1)*error+eps)^2;                          %set at square of uncertainty in omega1
P(2,2)=(omega2(1)*error+eps)^2;                          %set at square of uncertainty in omega2
P(3,3)=(omega3(1)*error+eps)^2;                          %set at square of uncertainty in omega3
P(4,4)=(q1(1)*error+eps)^2;                              %set at square of uncertainty in q1
P(5,5)=(q2(1)*error+eps)^2;                              %set at square of uncertainty in q2
P(6,6)=(q3(1)*error+eps)^2;                              %set at square of uncertainty in q3
P(7,7)=(q4(1)*error+eps)^2;                              %set at square of uncertainty in q4

else

%...or can manually enter whatever intialization values you wish...

omega1hat(1)=input('Please enter initial omega1 in rpm:  ');     %apriori estimate of omega1
omega2hat(1)=input('Please enter initial omega2 in rpm:  ');     %apriori estimate of omega2
omega3hat(1)=input('Please enter initial omega3 in rpm:  ');     %apriori estimate of omega3
q1hat(1)=input('Please enter initial q1:  ');                    %apriori estimate of q1
q2hat(1)=input('Please enter initial q2:  ');                    %apriori estimate of q2
q3hat(1)=input('Please enter initial q3:  ');                    %apriori estimate of q3
q4hat(1)=input('Please enter initial q4:  ');                    %apriori estimate of q4

omega1hat(1)=omega1hat(1)*2*pi/60;
omega2hat(1)=omega2hat(1)*2*pi/60;
omega3hat(1)=omega3hat(1)*2*pi/60;                               %convert manual omega entries to rad/s

'For manual entry of initial conditions, estimated uncertainty in the initial conditions'
'is necessary to intialize the covariance matrix...'
' '

domega1hat=input('Please enter estimated uncertainty in initial omega1 (rad/s):  ');     %uncertainty in initial estimate of omega1
domega2hat=input('Please enter estimated uncertainty in initial omega2 (rad/s):  ');     %uncertainty in initial estimate of omega2
domega3hat=input('Please enter estimated uncertainty in initial omega3 (rad/s):  ');     %uncertainty in initial estimate of omega3
dq1hat=input('Please enter estimated uncertainty in initial q1:  ');                     %uncertainty in initial estimate of q1
dq2hat=input('Please enter estimated uncertainty in initial q2:  ');                     %uncertainty in initial estimate of q2
dq3hat=input('Please enter estimated uncertainty in initial q3:  ');                     %uncertainty in initial estimate of q3
dq4hat=input('Please enter estimated uncertainty in initial q4:  ');                     %uncertainty in initial estimate of q4

%***** Non-zero entries of initial covariance matrix for manual initial cond *****

P(1,1)=(domega1hat+eps)^2;                    %set at square of uncertainty in initial estimate of omega1
P(2,2)=(domega2hat+eps)^2;                    %set at square of uncertainty in initial estimate of omega2
P(3,3)=(domega3hat+eps)^2;                    %set at square of uncertainty in initial estimate of omega3
P(4,4)=(dq1hat+eps)^2;                        %set at square of uncertainty in initial estimate of q1
P(5,5)=(dq2hat+eps)^2;                        %set at square of uncertainty in initial estimate of q2
P(6,6)=(dq3hat+eps)^2;                        %set at square of uncertainty in initial estimate of q3
P(7,7)=(dq4hat+eps)^2;                        %set at square of uncertainty in initial estimate of q4
```

```
end

%insure initial quaternion estimate is a unit quaternion

mag=sqrt(q1hat(1)^2+q2hat(1)^2+q3hat(1)^2+q4hat(1)^2);
q1hat(1)=q1hat(1)/mag;
q2hat(1)=q2hat(1)/mag;
q3hat(1)=q3hat(1)/mag;
q4hat(1)=q4hat(1)/mag;

%***** Enter measurement noise values for use in calculating Measurement Noise Matrix, R *****

'Do you wish to enter sensor measurement sigmas manually or let the program use the actual'
'sensor measurement sigmas with a user defined percent error to make an initial estimate'
'(default is to let the program do it)?'
' '
sigmas=input('Enter "1" for manual entry and "0" for automatic:  ');
if isempty(sigmas)
  sigmas=0;
end
clc
if sigmas == 0

%***** base sensor noise matrix on actual sigmas plus user defined error *****

sensorerror=input('Enter percentage error (+/-) in sensor measurement sigmas (i.e., -5.2% = "-5.2") Default is 0%:  ');
if isempty(sensorerror)
  sensorerror=0;
end

'percent error applied to the actual sensor measurement sigmas is: ',sensorerror

sensorerror=sensorerror/100;

SIGOMEGA1=stdomega1;                    %right now...calculated in attitude_filter from std of actual error
SIGOMEGA2=stdomega2;                    %right now...calculated in attitude_filter from std of actual error
SIGOMEGA3=stdomega3;                    %right now...calculated in attitude_filter from std of actual error
estSIGOMEGA1=SIGOMEGA1+SIGOMEGA1*sensorerror;
estSIGOMEGA2=SIGOMEGA2+SIGOMEGA2*sensorerror;
estSIGOMEGA3=SIGOMEGA3+SIGOMEGA3*sensorerror;
estSIGSUNX=SIGSUNX+SIGSUNX*sensorerror;
estSIGSUNY=SIGSUNY+SIGSUNY*sensorerror;
estSIGSUNZ=SIGSUNZ+SIGSUNZ*sensorerror;
```

```
estSIGMAGX=SIGMAGX+SIGMAGX*sensorerror;
estSIGMAGY=SIGMAGY+SIGMAGY*sensorerror;
estSIGMAGZ=SIGMAGZ+SIGMAGZ*sensorerror;

'apriori estimate of omega1 "measurement" sigma (rad/s): ',estSIGOMEGA1                    %apriori estimate of SIGOMEGA1
'apriori estimate of omega2 "measurement" sigma (rad/s): ',estSIGOMEGA2                    %apriori estimate of SIGOMEGA2
'apriori estimate of omega3 "measurement" sigma (rad/s): ',estSIGOMEGA3                    %apriori estimate of SIGOMEGA3
'apriori estimate of sun sensor x axis measurement sigma (deg): ',estSIGSUNX*180/pi         %apriori estimate of SIGSUNX
'apriori estimate of sun sensor y axis measurement sigma (deg): ',estSIGSUNY*180/pi         %apriori estimate of SIGSUNY
'apriori estimate of sun sensor z axis measurement sigma (deg): ',estSIGSUNZ*180/pi         %apriori estimate of SIGSUNZ
'apriori estimate of mag field sensor x axis measurement sigma (deg): ',estSIGMAGX*180/pi   %apriori estimate of SIGMAGX
'apriori estimate of mag field sensor y axis measurement sigma (deg): ',estSIGMAGY*180/pi   %apriori estimate of SIGMAGY
'apriori estimate of mag field sensor z axis measurement sigma (deg): ',estSIGMAGZ*180/pi   %apriori estimate of SIGMAGZ

else

%...or can manually enter whatever intialization values you wish...

estSIGOMEGA1=input('Please enter sigma for the omega1 derived "measurement" in rpm:  ');  %apriori estimate of SIGOMEGA1
estSIGOMEGA2=input('Please enter sigma for the omega2 derived "measurement" in rpm:  ');  %apriori estimate of SIGOMEGA2
estSIGOMEGA3=input('Please enter sigma for the omega3 derived "measurement" in rpm:  ');  %apriori estimate of SIGOMEGA3

estSIGOMEGA1=estSIGOMEGA1*2*pi/60;                                                        %convert estSIGOMEGA1 to rad/s
estSIGOMEGA2=estSIGOMEGA2*2*pi/60;                                                        %convert estSIGOMEGA2 to rad/s
estSIGOMEGA3=estSIGOMEGA3*2*pi/60;                                                        %convert estSIGOMEGA3 to rad/s

estSIGSUNX=input('Please enter sigma for the sun sensor x axis measurement in deg:  ');         %apriori estimate of SIGSUNX
estSIGSUNY=input('Please enter sigma for the sun sensor y axis measurement in deg:  ');         %apriori estimate of SIGSUNY
estSIGSUNZ=input('Please enter sigma for the sun sensor z axis measurement in deg:  ');         %apriori estimate of SIGSUNZ
estSIGMAGX=input('Please enter sigma for the mag field sensor x axis measurement in deg:  ');  %apriori estimate of SIGMAGX
estSIGMAGY=input('Please enter sigma for the mag field sensor y axis measurement in deg:  ');  %apriori estimate of SIGMAGY
estSIGMAGZ=input('Please enter sigma for the mag field sensor z axis measurement in deg:  ');  %apriori estimate of SIGMAGZ

estSIGSUNX=estSIGSUNX*pi/180;                           %convert from degrees to radians
estSIGSUNY=estSIGSUNY*pi/180;                           %convert from degrees to radians
estSIGSUNZ=estSIGSUNZ*pi/180;                           %convert from degrees to radians
estSIGMAGX=estSIGMAGX*pi/180;                           %convert from degrees to radians
estSIGMAGY=estSIGMAGY*pi/180;                           %convert from degrees to radians
estSIGMAGZ=estSIGMAGZ*pi/180;                           %convert from degrees to radians

end

'Please press "return" to continue...'
pause;clc;close;
```

```
%***** Non-zero submatrices of Measurement Noise Matrix, R *****

ROMEGA=diag([estSIGOMEGA1^2+eps  estSIGOMEGA2^2+eps   estSIGOMEGA3^2+eps]);
RSENSOR=diag([estSIGSUNX^2+eps estSIGSUNY^2+eps   estSIGSUNZ^2+eps   estSIGMAGX^2+eps   estSIGMAGY^2+eps
estSIGMAGZ^2+eps]);

%***** user input of process noise to reflect uncertainty in dynamic model, PHIS *****
''
confidence=input('Enter process noise to reflect uncertainty in model (default is 1.07):  ');

if isempty(confidence)
  confidence=1.07;
end

PHIS=confidence;          %(100-confidence)/100;                                          %[17] pg. 317

'Process noise applied is: ',PHIS

for i=1:1:statesize;
  Q(i,i)=PHIS^2;
end
''
%***** time constants and noise variances for noise driving motion *****

'enter time constants associated with motion...'
''

tauomega1 = input('enter tau omega1 (default=.07)    ');
if isempty(tauomega1)
   tauomega1=.07
end
tauomega2 = input('enter tau omega2 (default = .07)    ');
if isempty(tauomega2)
   tauomega2=.07
end
tauomega3 = input('enter tau omega3 (default=1e7)    ');
if isempty(tauomega3)
   tauomega3=1e7
end

'press "return" to continue...'
pause;clc;close;
```

```
'Enter the scaling parameters for the Unscented Transform...'
' '
alpha=input('enter alpha (default is .01):  ');                  %determines spread of sigma values about mean, .0001<alpha<1
if isempty (alpha)
  alpha=.01
end
' '
beta=input('enter beta (default is 2):  ');                            %incorporates prior knowledge of state dist, beta=2
for Gaussian
if isempty (beta)
  beta=2
end
' '
augalpha=input('enter alpha for augmented sigma (default is .01):  ');
if isempty (augalpha)
  augalpha=.01
end
                                                                  %determines spread of sigma values about mean, .0001<alpha<1
' '
augbeta=input('enter beta for augmented sigma (default is 2):  ');
if isempty (augbeta)
  augbeta=2
end
                                                                  %incorporates prior knowledge of state dist, beta=2 for Gaussian

' '
'Please wait while data is processed...'

%***** loop to calculate estimated states at each time step *****

count=1;
                                                                  %initialize counter
t=0.;
                                                                  %initialize time
ArrayT(1)=t;

while count<=length(omega1_q_min)-1

  %***** Assign variables to more managable variable names *****

  x1=omega1hat(count);
  x2=omega2hat(count);
  x3=omega3hat(count);
  x4=q1hat(count);
```

```
x5=q2hat(count);
x6=q3hat(count);
x7=q4hat(count);

den=sqrt(x4^2+x5^2+x6^2+x7^2);          %convenient compilation of terms to be used later
den3=den^3;                             %convenient compilation of terms to be used later
f4=(x1*x7-x2*x6+x3*x5);                 %convenient compilation of terms to be used later
f5=(x1*x6+x2*x7-x3*x4);                 %convenient compilation of terms to be used later
f6=(-x1*x5+x2*x4+x3*x7);                %convenient compilation of terms to be used later
f7=(-x1*x4-x2*x5-x3*x6);                %convenient compilation of terms to be used later

%***** Current estimate of the state vector *****

eststate=[x1 x2 x3 x4 x5 x6 x7]';

%***** Calculate parameter values *****

L=statesize;                            %dimension of state vector

kappa=3-L;                              %secondary scaling parameter
lambda=alpha^2*(L+kappa)-L;             %primary scaling parameter
gamma=sqrt(L+lambda);

if isempty(find(eig(P)<0))              %check for positive-definiteness of P
  C=chol(P);
else
  C=zeros(statesize,statesize);         %Cholesky factorization of covariance matrix
end

%***** Calculate Sigma Point Matrix, sigma *****

sigma=[ ];
sigma(:,1)=eststate;                    %Sigma-zero
for i=2:1:L+1
  sigma(:,i)=sigma(:,1)+gamma*C(i-1,:)'; %Sigma points 1 through L are eststate plus gamma times the
                                         %transpose of the ith row of the Cholesky upper triangular
                                         %factorization
end

for i=L+2:1:2*L+1
  sigma(:,i)=sigma(:,1)-gamma*C(i-(L+1),:)';        %Sigma points L+1 through 2L are eststate minus gamma
                                                    %times the transpose of the ith row of the Cholesky upper
                                                    %triangular factorization
end
```

```
%***** Instantiate Sigma Points Through Nonlinear Function, sigmabar *****
%***** Propagate estimated states forward using one-step Euler integration *****

sigmabar=[ ];

for j=1:1:length(sigma(1,:))

  X1=sigma(1,j);                        %omega1hat component to be propagated forward one time step
  X2=sigma(2,j);                        %omega2hat component to be propagated forward one time step
  X3=sigma(3,j);                        %omega3hat component to be propagated forward one time step
  X4=sigma(4,j);                        %q1hat component to be propagated forward one time step
  X5=sigma(5,j);                        %q2hat component to be propagated forward one time step
  X6=sigma(6,j);                        %q3hat component to be propagated forward one time step
  X7=sigma(7,j);                        %q4hat component to be propagated forward one time step

S=0;                                    %initialize integration timer
H1=.01;                                 %set step size for numerical integration

while S<=(TS-.0001)                     %loops from current step to just short of next time step

  X1dot=-1/tauomega1*X1;
  X1=X1+H1*X1dot;

  X2dot=-1/tauomega2*X2;
  X2=X2+H1*X2dot;

  X3dot=-1/tauomega3*X3;
  X3=X3+H1*X3dot;

  X4dot=.5*f4/den;
  X4=X4+H1*X4dot;

  X5dot=.5*f5/den;
  X5=X5+H1*X5dot;

  X6dot=.5*f6/den;
  X6=X6+H1*X6dot;

  X7dot=.5*f7/den;
  X7=X7+H1*X7dot;

  S=S+H1;                               %increment integration timer by defined integration step size
end                                     %yields each state estimate integrated forward to next time step
```

```
sigmabar(1,j)=X1;                                    %transformed sigma points
sigmabar(2,j)=X2;
sigmabar(3,j)=X3;
sigmabar(4,j)=X4;
sigmabar(5,j)=X5;
sigmabar(6,j)=X6;
sigmabar(7,j)=X7;

end

%***** Compute the predicted mean of each state *****

Wm0=lambda/(L+lambda);                               %weight Wm for initial step
Wm=1/(2*(L+lambda));                                 %Wm for steps other than zero

X1bar=Wm0*sigmabar(1,1);
X2bar=Wm0*sigmabar(2,1);
X3bar=Wm0*sigmabar(3,1);
X4bar=Wm0*sigmabar(4,1);                             %this block predicts mean as a weighted sum
X5bar=Wm0*sigmabar(5,1);
X6bar=Wm0*sigmabar(6,1);
X7bar=Wm0*sigmabar(7,1);

for k=2:1:2*L+1
   X1bar=X1bar+Wm*sigmabar(1,k);
   X2bar=X2bar+Wm*sigmabar(2,k);
   X3bar=X3bar+Wm*sigmabar(3,k);
   X4bar=X4bar+Wm*sigmabar(4,k);
   X5bar=X5bar+Wm*sigmabar(5,k);
   X6bar=X6bar+Wm*sigmabar(6,k);
   X7bar=X7bar+Wm*sigmabar(7,k);
end

statebar=[X1bar X2bar X3bar X4bar X5bar X6bar X7bar]';    %vector of projected states

%***** Compute the predicted covariance before the update, M *****

Wc0=lambda/(L+lambda)+1-alpha^2+beta;                %weight Wc for initial step
Wc=1/(2*(L+lambda));                                 %Wc for steps other than zero

M=Wc0*((sigmabar(:,1)-statebar)*(sigmabar(:,1)-statebar)');

for m=2:1:2*L+1
```

```
   M=M+Wc*((sigmabar(:,m)-statebar)*(sigmabar(:,m)-statebar)');
 end

 %***** Augment Sigma point Matrix to Account for Process Noise Covariance, Q *****

 Lprime=2*L;                                        %augment with additional points
 augkappa=3-Lprime;                                 %secondary scaling parameter
 auglambda=augalpha^2*(Lprime+augkappa)-Lprime;     %primary scaling parameter
 auggamma=sqrt(Lprime+auglambda);                   %gamma for augmented matrix
 if isempty(find(eig(Q)<0))                         %check for positive-definiteness
   C2=chol(Q);
 else
   C2=zeros(statesize,statesize);                   %Cholesky factorization of process noise covariance matrix
 end

 newsigma=sigmabar;
 for n=1:1:statesize
   newsigma(:,2*L+1+n)=sigmabar(:,1)+auggamma*C2(n,:)';
 end

 for n=statesize+1:1:2*statesize
   newsigma(:,2*L+1+n)=sigmabar(:,1)-auggamma*C2(n-L,:)';
 end

%***** Instantiate transformed Sigma points through non-linear observation model *****
 %***** In this implementation, the observations are the states (zetabar=newsigma)!*****

 for p=1:1:length(newsigma(1,:))
   zetabar(1,p)=newsigma(1,p);
   zetabar(2,p)=newsigma(2,p);
   zetabar(3,p)=newsigma(3,p);
   zetabar(4,p)=newsigma(4,p);
   zetabar(5,p)=newsigma(5,p);
   zetabar(6,p)=newsigma(6,p);
   zetabar(7,p)=newsigma(7,p);
 end

 %***** Compute the predicted observation vector, obsbar *****

 augWm0=auglambda/(Lprime+auglambda);               %weight Wm for initial step
 augWm=1/(2*(Lprime+auglambda));                    %Wm for steps other than zero

 z1bar=augWm0*zetabar(1,1);
 z2bar=augWm0*zetabar(2,1);
```

```
z3bar=augWm0*zetabar(3,1);
z4bar=augWm0*zetabar(4,1);                          %this block predicts mean as a weighted sum
z5bar=augWm0*zetabar(5,1);
z6bar=augWm0*zetabar(6,1);
z7bar=augWm0*zetabar(7,1);

for k=2:1:length(newsigma(1,:))
  z1bar=z1bar+augWm*zetabar(1,k);
  z2bar=z2bar+augWm*zetabar(2,k);
  z3bar=z3bar+augWm*zetabar(3,k);
  z4bar=z4bar+augWm*zetabar(4,k);
  z5bar=z5bar+augWm*zetabar(5,k);
  z6bar=z6bar+augWm*zetabar(6,k);
  z7bar=z7bar+augWm*zetabar(7,k);
end

obsbar=[z1bar z2bar z3bar z4bar z5bar z6bar z7bar]';          %vector of projected observations

%***** Calculate Measurement Noise Matrix, RMAT *****
%***** Uses A matrix from Gauss_newton to relate 6 x 6 RSENSOR to covariance of quaternions after convergence [11] ******

RMAT=[ROMEGA, zeros(3,4);zeros(4,3), A(:,:,count)*RSENSOR*A(:,:,count)'];

%***** Calculate Innovation Covariance Matrix, Pzz *****

augWc0=auglambda/(Lprime+auglambda)+1-augalpha^2+augbeta;  %weight Wc for initial step
augWc=1/(2*(Lprime+auglambda));                            %Wc for steps other than zero
Pzz=augWc0*((zetabar(:,1)-obsbar)*(zetabar(:,1)-obsbar)');
for r=2:1:2*Lprime+1
  Pzz=Pzz+augWc*((zetabar(:,r)-obsbar)*(zetabar(:,r)-obsbar)');
end

Pzz=Pzz+RMAT;

%***** Calculate Cross-Correlation Matrix, Pxz *****

Pxz=augWc0*((newsigma(:,1)-statebar)*(zetabar(:,1)-obsbar)');

for r=2:1:2*Lprime+1
  Pxz=Pxz+augWc*((newsigma(:,r)-statebar)*(zetabar(:,r)-obsbar)');
end

%***** Calculate Kalman Gain Matrix *****
```

```
K=Pxz*inv(Pzz);                                    %calculate Kalman gain matrix

%***** Form observation vector (measurement) *****

obs(1:7,count+1)=[omega1_q_min(count+1) omega2_q_min(count+1) omega3_q_min(count+1) q_min(1,count+1)
q_min(2,count+1) q_min(3,count+1) q_min(4,count+1)]';

%***** Calculate updated state estimates using propagated states, residuals and Kalman gains

statehat=statebar-K*(obsbar-obs(1:7,count+1));

%***** Calculate Covariance After the Update *****

P=M-K*Pzz*K';                                      %calcuate covariance after update
P=sqrt(P.^2);

%***** Increment time *****

t=t+TS;                                            %increment time

%***** Check for loss of data *****

if q_min(1:4,count)==[0 0 0 1]';

  omega1hat(count+1)=omega1hat(count);
  omega2hat(count+1)=omega2hat(count);
  omega3hat(count+1)=omega3hat(count);
  q1hat(count+1)=q1hat(count);
  q2hat(count+1)=q2hat(count);
  q3hat(count+1)=q3hat(count);
  q4hat(count+1)=q4hat(count);
  flag(count)=1;

else

  %*************** Data management *****************
  omega1hat(count+1)=statehat(1);
  omega2hat(count+1)=statehat(2);
  omega3hat(count+1)=statehat(3);
  q1hat(count+1)=statehat(4);
  q2hat(count+1)=statehat(5);
  q3hat(count+1)=statehat(6);
  q4hat(count+1)=statehat(7);
end
```

```
    ArrayT(count+1)=t;

    %*************** Error Analysis ******************

    SP11=sqrt(P(1,1));                          %Calculate theoretical upper error bound on omega1hat
    SP11P=-SP11;                                %Calculate theoretical lower error bound on omega1hat
    SP22=sqrt(P(2,2));                          %Calculate theoretical upper error bound on omega2hat
    SP22P=-SP22;                                %Calculate theoretical lower error bound on omega2hat
    SP33=sqrt(P(3,3));                          %Calculate theoretical upper error bound on omega3hat
    SP33P=-SP33;                                %Calculate theoretical lower error bound on omega3hat
    SP44=sqrt(P(4,4));                          %Calculate theoretical upper error bound on q1hat
    SP44P=-SP44;                                %Calculate theoretical lower error bound on q1hat
    SP55=sqrt(P(5,5));                          %Calculate theoretical upper error bound on q2hat
    SP55P=-SP55;                                %Calculate theoretical lower error bound on q2hat
    SP66=sqrt(P(6,6));                          %Calculate theoretical upper error bound on q3hat
    SP66P=-SP66;                                %Calculate theoretical lower error bound on q3hat
    SP77=sqrt(P(7,7));                          %Calculate theoretical upper error bound on q4hat
    SP77P=-SP77;                                %Calculate theoretical lower error bound on q4hat

    ArraySP11(count+1)=SP11;
    ArraySP11P(count+1)=SP11P;
    ArraySP22(count+1)=SP22;
    ArraySP22P(count+1)=SP22P;
    ArraySP33(count+1)=SP33;
    ArraySP33P(count+1)=SP33P;
    ArraySP44(count+1)=SP44;
    ArraySP44P(count+1)=SP44P;
    ArraySP55(count+1)=SP55;
    ArraySP55P(count+1)=SP55P;
    ArraySP66(count+1)=SP66;
    ArraySP66P(count+1)=SP66P;
    ArraySP77(count+1)=SP77;
    ArraySP77P(count+1)=SP77P;

    count=count+1;                              %increment counter

end

%****** calculate mean square errors for "instantaneous" and for UKF ******
%****** mean square error excludes first 5 samples to reduce effects of transients *****

s2=length(omega1_q_min);
s1=5;
```

```
omega1_q_min_mse=mean_square_error(omega1_q_min(s1:s2),omega1(s1:s2));
omega2_q_min_mse=mean_square_error(omega2_q_min(s1:s2),omega2(s1:s2));
omega3_q_min_mse=mean_square_error(omega3_q_min(s1:s2),omega3(s1:s2));
q1_q_min_mse=mean_square_error(q_min(1,(s1:s2)),q1(s1:s2));
q2_q_min_mse=mean_square_error(q_min(2,(s1:s2)),q2(s1:s2));
q3_q_min_mse=mean_square_error(q_min(3,(s1:s2)),q3(s1:s2));
q4_q_min_mse=mean_square_error(q_min(4,(s1:s2)),q4(s1:s2));

omega1_ukf_mse=mean_square_error(omega1hat(s1:s2),omega1(s1:s2));
omega2_ukf_mse=mean_square_error(omega2hat(s1:s2),omega2(s1:s2));
omega3_ukf_mse=mean_square_error(omega3hat(s1:s2),omega3(s1:s2));
q1_ukf_mse=mean_square_error(q1hat(s1:s2),q1(s1:s2));
q2_ukf_mse=mean_square_error(q2hat(s1:s2),q2(s1:s2));
q3_ukf_mse=mean_square_error(q3hat(s1:s2),q3(s1:s2));
q4_ukf_mse=mean_square_error(q4hat(s1:s2),q4(s1:s2));

%*************** Plotting and output statements *****************

plots=input('Enter "1" to display plots. Enter "0" for no plots. Default is no plots:  ');
if isempty(plots)
  plots=0;
end

if plots==1

%****** plots of error vs. true and theoretical error bounds *******

plot(ArrayT,(omega1(1:s2)-omega1hat),ArrayT,ArraySP11,'x',ArrayT,ArraySP11P,'x');
grid on;title('error in EKF estimate of omega1');
legend('error in omega1hat','theoretical bound');
xlabel('time (sec)');ylabel('error in omega1hat (rad/s)');
pause;close;clc;

plot(ArrayT,(omega2(1:s2)-omega2hat),ArrayT,ArraySP22,'x',ArrayT,ArraySP22P,'x');
grid on;title('error in EKF estimate of omega2');
legend('error in omega2hat','theoretical bound');
xlabel('time (sec)');ylabel('error in omega2hat (rad/s)');
pause;close;clc;

plot(ArrayT,(omega3(1:s2)-omega3hat),ArrayT,ArraySP33,'x',ArrayT,ArraySP33P,'x');
grid on;title('error in EKF estimate of omega3');
legend('error in omega3hat','theoretical bound');
xlabel('time (sec)');ylabel('error in omega3hat (rad/s)');
```

```
pause;close;clc;

plot(ArrayT,(q1(1:s2)-q1hat),ArrayT,ArraySP44,'x',ArrayT,ArraySP44P,'x');
grid on;title('error in EKF estimate of q1');
legend('error in q1hat','theoretical bound');
xlabel('time (sec)');ylabel('error in q1');
pause;close;clc;

plot(ArrayT,(q2(1:s2)-q2hat),ArrayT,ArraySP55,'x',ArrayT,ArraySP55P,'x');
grid on;title('error in EKF estimate of q2');
legend('error in q2hat','theoretical bound');
xlabel('time (sec)');ylabel('error in q2');
pause;close;clc;

plot(ArrayT,(q3(1:s2)-q3hat),ArrayT,ArraySP66,'x',ArrayT,ArraySP66P,'x');
grid on;title('error in UKF estimate of q3');
legend('error in q3hat','theoretical bound');
xlabel('time (sec)');ylabel('error in q3');
pause;close;clc;

plot(ArrayT,(q4(1:s2)-q4hat),ArrayT,ArraySP77,'x',ArrayT,ArraySP77P,'x');
grid on;title('error in UKF estimate of q4');
legend('error in q4hat','theoretical bound');
xlabel('time (sec)');ylabel('error in q4');
pause;close;clc;

%****** plots to show true vs. quaternion derived alone vs. UKF estimates *******

plot( ArrayT, omega1_q_min,'gx',ArrayT, omega1hat,'m', ArrayT, omega1(1:s2),'--');grid on;
title('Quaternion derived omega1, UKF estimate of omega1, "True" omega1' );
xlabel('time (sec)');ylabel('omega1 (rad/s)');
legend('omega1 from differencing quaternions','omega1 from UKF','true omega1');
pause;clc;close;

plot( ArrayT, omega2_q_min,'gx',ArrayT, omega2hat,'m', ArrayT, omega2(1:s2),'--');grid on;
title('Quaternion derived omega2, UKF estimate of omega2, "True" omega2' );
xlabel('time (sec)');ylabel('omega2 (rad/s)');
legend('omega2 from differencing quaternions','omega2 from UKF','true omega2');
pause;clc;close;

plot( ArrayT, omega3_q_min,'gx',ArrayT, omega3hat,'m', ArrayT, omega3(1:s2),'--');grid on;
title('Quaternion derived omega3, UKF estimate of omega3, "True" omega3' );
xlabel('time (sec)');ylabel('omega3 (rad/s)');
legend('omega3 from differencing quaternions','omega3 from UKF','true omega3');
```

```
pause;clc;close;

end

'Summary of performance of each method...'
' '
'The MSE in omega1 derived from differencing quaternions is (rad/s): ',' ',omega1_q_min_mse
' '
'The MSE in omega1 for the UKF estimates is (rad/s): ',' ',omega1_ukf_mse
' '
'The MSE in omega2 derived from differencing quaternions is (rad/s): ',' ',omega2_q_min_mse
' '
'The MSE in omega2 for the UKF estimates is (rad/s): ',' ',omega2_ukf_mse
' '
'Please press "return" to continue...'
pause;clc;close;

'The MSE in omega3 derived from differencing quaternions is (rad/s): ',' ',omega3_q_min_mse
' '
'The MSE in omega3 for the UKF estimates is (rad/s): ',' ',omega3_ukf_mse
' '
'The MSE in q1 derived using Gauss-Newton is: ',' ',q1_q_min_mse
' '
'The MSE in q1 for the UKF estimates is: ',' ',q1_ukf_mse
' '
'Please press "return" to continue...'
pause;clc;close;

'Summary of performance of each method (cont.) ...'
' '
'The MSE in q2 derived using Gauss-Newton is: ',' ',q2_q_min_mse
' '
'The MSE in q2 for the UKF estimates is: ',' ',q2_ukf_mse
' '
'press "return" to continue...'
pause;clc;close;

'Summary of performance of each method (cont.) ...'
' '
'The MSE in q3 derived using Gauss-Newton is: ',' ',q3_q_min_mse
' '
'The MSE in q3 for the UKF estimates is: ',' ',q3_ukf_mse
' '
'The MSE in q4 derived using Gauss-Newton is: ',' ',q4_q_min_mse
```

```
' '
'The MSE in q4 for the UKF estimates is: ',' ',q4_ukf_mse
' '
'press "return" to continue...'
pause;clc;close;

'Summary of performance of each method (cont.) ...'

%****** rotate each vector using actual and UKF attitude quaternion ******
%****** check mean-square-error between vectors ******

'****** Evaluate Mean-Square-Error of rotated vectors using Gauss-Newton derived quaternion ******'
'Press "return" to rotate the reference vector at each time step to the'
'representation in the rotated frame using both the actual rotation quaternion'
'and the quaternion derived using the UKF...once rotation is complete,'
'calculate the mean-square-error between the results.'
pause;close;clc;

qukf=[q1hat;q2hat;q3hat;q4hat];                              %matrix representation of ukf quaternions

ukf_mse_error=[ ];
for j=1:count;
   ukfvec(1:6,j)=qframerot6(qukf(1:4,j),vref(1:6,j));        %rotate ref vector using min error quaternion
   Q_vec(1:6,j)=qframerot6(Qvec(1:4,j),vref(1:6,j));         %rotate ref vector using true quaternion
   ukf_mse_error(j)=mean_square_error(ukfvec(1:6,j),Q_vec(1:6,j)); %calculate MSE between rotated vectors
end

%***** determine angle between rotated vectors *****
for k=1:count
   ukfsunangle(k)=acos(dot(ukfvec(1:3,k),Q_vec(1:3,k))/(norm(ukfvec(1:3,k))*norm(Q_vec(1:3,k))));
   ukfmagangle(k)=acos(dot(ukfvec(4:6,k),Q_vec(4:6,k))/(norm(ukfvec(4:6,k))*norm(Q_vec(4:6,k))));
   GNsunangle(k)=acos(dot(q_minvec(1:3,k),Q_vec(1:3,k))/(norm(q_minvec(1:3,k))*norm(Q_vec(1:3,k))));
   GNmagangle(k)=acos(dot(q_minvec(4:6,k),Q_vec(4:6,k))/(norm(q_minvec(4:6,k))*norm(Q_vec(4:6,k))));
end
ukfsunangle=ukfsunangle*180/pi;
ukfmagangle=ukfmagangle*180/pi;
GNsunangle=GNsunangle*180/pi;
GNmagangle=GNmagangle*180/pi;

%****** Plot vectors rotated using derived and actual quaternions *****
%plot3(Q_vec(1,1:count),Q_vec(2,1:count),Q_vec(3,1:count),'b');grid on;hold on;
%plot3(ukfvec(1,1:count),ukfvec(2,1:count),ukfvec(3,1:count),'m');
%plot3(Q_vec(4,1:count),Q_vec(5,1:count),Q_vec(6,1:count),'r');grid on;hold on;
%plot3(ukfvec(4,1:count),ukfvec(5,1:count),ukfvec(6,1:count),'g');
```

```
%xlabel('X component');ylabel('Y component');zlabel('Z component');
%legend('sun vector, actual q','sun vector, ukf q','mag field vector, actual q', 'mag field vector, ukf q');
%title('3-D plot of reference vectors rotated using actual quaternions and ukf derived quaternions');
%pause;close;clc;

%****** Plot mean-square-error between reference vector rotated using actual quaternion ******
%****** and reference vector rotated using derived quaternion.                    ******

%plot(T(1:count),ukf_mse_error(1:count)),grid on;
%title('mean-square error between reference vectors rotated using ukf derived and actual quaternions');
%xlabel('time (sec)');ylabel('MSE');legend('MSE')

%'press "return" to continue...'
%pause;close;clc;

'********************* Overall Performance Indicator ************************'
'****** Excludes first five measurements to reduce effect of transients ******'
' '
'The average of the omega mean-square-errors for the differencing method is:  '
GNomegaperf=(omega1_q_min_mse+omega2_q_min_mse+omega3_q_min_mse)/3
' '
'The average of the omega mean-square-errors for the ukf results is:  '
ukfomegaperf=(omega1_ukf_mse+omega2_ukf_mse+omega3_ukf_mse)/3
' '
'The average of the quaternion component mean-square-errors for the Gauss-Newton method is:  '
GNqperf=(q1_q_min_mse+q2_q_min_mse+q3_q_min_mse+q4_q_min_mse)/4
' '
'The average of the quaternion component mean-square-errors for the ukf results is:  '
ukfqperf=(q1_ukf_mse+q2_ukf_mse+q3_ukf_mse+q4_ukf_mse)/4
' '
'press "return" to continue...'
pause;clc;close;

'Summary of performance of each method (cont.) ...'
' '
'The average MSE for the ref vectors rotated using GN quaternions:  '
mean_GN_mse_error=mean(GN_mse_error)
' '
'The average MSE for the ref vectors rotated using ukf quaternions:  '
mean_ukf_mse_error=mean(ukf_mse_error)
' '
'press "return" to continue...'
pause;clc;close;
```

```
'Summary of performance of each method (cont.) ...'
' '
'overall GN performance: sqrt[(mean omega MSE)^2 + (mean rotated vector MSE)^2)]'
overall_GN_perf=sqrt(GNomegaperf^2+mean_GN_mse_error^2)
' '
'overall ukf performance: sqrt[(mean omega MSE)^2 + (mean rotated vector MSE)^2)]'
overall_ukf_perf=sqrt(ukfomegaperf^2+mean_ukf_mse_error^2)
' '
%'******************************************************************************************'
%'********************************* ukf summary ****************************************'
%'******************************************************************************************'
'omega:      ',ukfomegaperf
'quaternion:  ',mean_ukf_mse_error
'overall:     ',overall_ukf_perf
'mean sun angle error:    ',mean(ukfsunangle)
'std sun angle error:    ',std(ukfsunangle)
'max sun angle error:    ',max(ukfsunangle)
'mean mag angle error:    ',mean(ukfmagangle)
'std mag angle error:    ',std(ukfmagangle)
'max mag angle error:    ',max(ukfmagangle)
' '
'press "return" to continue...'
pause;close;clc;

angles=input('Enter "1" to calculate attitude error angles. Enter "0" for no angles. Default is no angles:  ');
if isempty(angles)
  angles=0;
end

if angles==1

   %Calculate angles between body frame rotated using estimated quaternion and
   %rotated using "true" quaternion

   [ovec1,ovec2,ovec3,ovec1hat,ovec2hat,ovec3hat,ang1,ang2,ang3]=orientation3(Qvec, qukf);

end

%end of program
```

A.15 Rates_ekf.m

Rates_ekf.m is the MATLAB script that implements the extended Kalman filter (EKF) designed in Chapter 9. This implementation is for the case where rotational rates are directly measured by rate sensors and these noisy measurements are available to the filter. Rates_ekf.m is called by attitude_filter.m if the appropriate option is selected by the user.

```
%RATES_EKF - Discrete Extended Kalman Filter Implementation
%Uses data generated by attitude_truth.m to estimate 3-D body rates and
%the attitude quaternion of a sounding rocket given noisy measurements
%from two attitude sensors and noisy rate measuements from rate sensors.
%Uses a Gauss-Newton optimization and an Extended Kalman Filter (EKF) to
%generate a best estimate.
%
%Written by Mark Charlton.  Last modified 28 October 2003.

%Object subject to rotational motion -- non-linear Equation of Motion

%***** Initialize parameters and matrices *****

clc

ics=[ ];
sigmas=[ ];
error=[ ];                                              %initialize error multiplier
sensorerror=[ ];                                        %initialize error multiplier
ArrayT=[ ];                                             %initialize Arrays
q1hat=[ ];
q2hat=[ ];
q3hat=[ ];
q4hat=[ ];
omega1hat=[ ];
omega2hat=[ ];
omega3hat=[ ];
ArraySP11=[ ];
ArraySP11P=[ ];
ArraySP22=[ ];
ArraySP22P=[ ];
ArraySP33=[ ];
ArraySP33P=[ ];
ArraySP44=[ ];
ArraySP44P=[ ];
```

```
ArraySP55=[ ];
ArraySP55P=[ ];
ArraySP66=[ ];
ArraySP66P=[ ];
ArraySP77=[ ];
ArraySP77P=[ ];
H1=.01;                                   %assign integration step-size for Euler integration
statesize=7;                              %system order
PHIS=0;                                   %process noise (measure of uncertainty in dynamic model)
PHI=zeros(statesize,statesize);           %initialize state transition matrix, PHI
Q=zeros(statesize,statesize);             %initialize process noise matrix
P=zeros(statesize,statesize);             %initialize covariance matrix
HMAT=zeros(statesize,statesize);          %initialize linearized measurement matrix

%***** Assign initial estimates for omega1hat, omega2hat, omega3hat, q1hat, q2hat, q3hat, q4hat *****

'Do you wish to enter initial estimates of the rotational body rates and the quaternion components'
'or let the program use the actual initial conditions with a user defined percent error to'
'make an initial estimate (default is to let the program do it)?'
''
'If you choose manual entry, an uncertainty in the estimate is also required. The default is "auto".'
''
ics=input('Enter "1" for manual entry and "0" for automatic:  ');
if isempty(ics)
  ics=0;
end

if ics == 0

%***** base initialization on actual intial conditions with user defined standard error *****
clc
error=input('Enter percentage error (+/-) in initial conditions (i.e., -5.2% = "-5.2") Default is 5%:  ');
if isempty(error)
  error=5;
end

'percent error applied to the initial conditions is: ',error

error=error/100;

omega1hat(1)=omega1(1)+omega1(1)*error;

omega2hat(1)=omega2(1)+omega2(1)*error;
omega3hat(1)=omega3(1)+omega3(1)*error;
```

```
q1hat(1)=q1(1)+q1(1)*error;
q2hat(1)=q2(1)+q2(1)*error;
q3hat(1)=q3(1)+q3(1)*error;
q4hat(1)=q4(1)+q4(1)*error;

'apriori estimate of omega1 (rad/s): ',omega1hat(1)                    %apriori estimate of omega1
'apriori estimate of omega2 (rad/s): ',omega2hat(1)                    %apriori estimate of omega2
'apriori estimate of omega3 (rad/s): ',omega3hat(1)                    %apriori estimate of omega3
'apriori estimate of q1: ',q1hat(1)                            %apriori estimate of q1
'apriori estimate of q2: ',q2hat(1)                            %apriori estimate of q2
'apriori estimate of q3: ',q3hat(1)                            %apriori estimate of q3
'apriori estimate of q4: ',q4hat(1)                            %apriori estimate of q4

%***** Non-zero entries of initial covariance matrix for auto initial cond *****

P(1,1)=(omega1(1)*error+eps)^2;                                %set at square of uncertainty in omega1
P(2,2)=(omega2(1)*error+eps)^2;                                %set at square of uncertainty in omega2
P(3,3)=(omega3(1)*error+eps)^2;                                %set at square of uncertainty in omega3
P(4,4)=(q1(1)*error+eps)^2;                                    %set at square of uncertainty in q1
P(5,5)=(q2(1)*error+eps)^2;                                    %set at square of uncertainty in q2
P(6,6)=(q3(1)*error+eps)^2;                                    %set at square of uncertainty in q3
P(7,7)=(q4(1)*error+eps)^2;                                    %set at square of uncertainty in q4

else

%...or can manually enter whatever intialization values you wish...

omega1hat(1)=input('Please enter initial omega1 in rpm:  ');   %apriori estimate of omega1
omega2hat(1)=input('Please enter initial omega2 in rpm:  ');   %apriori estimate of omega2
omega3hat(1)=input('Please enter initial omega3 in rpm:  ');   %apriori estimate of omega3
q1hat(1)=input('Please enter initial q1:  ');                  %apriori estimate of q1
q2hat(1)=input('Please enter initial q2:  ');                  %apriori estimate of q2
q3hat(1)=input('Please enter initial q3:  ');                  %apriori estimate of q3
q4hat(1)=input('Please enter initial q4:  ');                  %apriori estimate of q4

omega1hat(1)=omega1hat(1)*2*pi/60;
omega2hat(1)=omega2hat(1)*2*pi/60;
omega3hat(1)=omega3hat(1)*2*pi/60;                             %convert manual omega entries to rad/s

'For manual entry of initial conditions, estimated uncertainty in the initial conditions'
'is necessary to intialize the covariance matrix...'
' '

domega1hat=input('Please enter estimated uncertainty in initial omega1 (rad/s):  ');     %uncertainty in initial estimate of omega1
domega2hat=input('Please enter estimated uncertainty in initial omega2 (rad/s):  ');     %uncertainty in initial estimate of omega2
```

```
domega3hat=input('Please enter estimated uncertainty in initial omega3 (rad/s):  ');    %uncertainty in initial estimate of omega3
dq1hat=input('Please enter estimated uncertainty in initial q1:  ');                   %uncertainty in initial estimate of q1
dq2hat=input('Please enter estimated uncertainty in initial q2:  ');                   %uncertainty in initial estimate of q2
dq3hat=input('Please enter estimated uncertainty in initial q3:  ');                   %uncertainty in initial estimate of q3
dq4hat=input('Please enter estimated uncertainty in initial q4:  ');                   %uncertainty in initial estimate of q4

%***** Non-zero entries of initial covariance matrix for manual initial cond *****

P(1,1)=(domega1hat+eps)^2;                    %set at square of uncertainty in initial estimate of omega1
P(2,2)=(domega2hat+eps)^2;                    %set at square of uncertainty in initial estimate of omega2
P(3,3)=(domega3hat+eps)^2;                    %set at square of uncertainty in initial estimate of omega3
P(4,4)=(dq1hat+eps)^2;                        %set at square of uncertainty in initial estimate of q1
P(5,5)=(dq2hat+eps)^2;                        %set at square of uncertainty in initial estimate of q2
P(6,6)=(dq3hat+eps)^2;                        %set at square of uncertainty in initial estimate of q3
P(7,7)=(dq4hat+eps)^2;                        %set at square of uncertainty in initial estimate of q4

end

%insure initial quaternion estimate is a unit quaternion

mag=sqrt(q1hat(1)^2+q2hat(1)^2+q3hat(1)^2+q4hat(1)^2);
q1hat(1)=q1hat(1)/mag;
q2hat(1)=q2hat(1)/mag;
q3hat(1)=q3hat(1)/mag;
q4hat(1)=q4hat(1)/mag;

%***** Enter measurement noise values for use in calculating Measurement Noise Matrix, R *****

'Do you wish to enter sensor measurement sigmas manually or let the program use the actual'
'sensor measurement sigmas with a user defined percent error to make an initial estimate'
'(default is to let the program do it)?'
' '
sigmas=input('Enter "1" for manual entry and "0" for automatic:  ');
if isempty(sigmas)
  sigmas=0;
end
clc
if sigmas == 0

%***** base sensor noise matrix on actual sigmas plus user defined error *****

sensorerror=input('Enter percentage error (+/-) in sensor measurement sigmas (i.e., -5.2% = "-5.2") Default is 0%:  ');
if isempty(sensorerror)
  sensorerror=0;
```

```
end

'percent error applied to the actual sensor measurement sigmas is: ',sensorerror

sensorerror=sensorerror/100;

SIGOMEGA1=stdomega1meas;                  %right now...calculated in attitude_filter from std of actual error
SIGOMEGA2=stdomega2meas;                  %right now...calculated in attitude_filter from std of actual error
SIGOMEGA3=stdomega3meas;                  %right now...calculated in attitude_filter from std of actual error
estSIGOMEGA1=SIGOMEGA1+SIGOMEGA1*sensorerror;
estSIGOMEGA2=SIGOMEGA2+SIGOMEGA2*sensorerror;
estSIGOMEGA3=SIGOMEGA3+SIGOMEGA3*sensorerror;
estSIGSUNX=SIGSUNX+SIGSUNX*sensorerror;
estSIGSUNY=SIGSUNY+SIGSUNY*sensorerror;
estSIGSUNZ=SIGSUNZ+SIGSUNZ*sensorerror;
estSIGMAGX=SIGMAGX+SIGMAGX*sensorerror;
estSIGMAGY=SIGMAGY+SIGMAGY*sensorerror;
estSIGMAGZ=SIGMAGZ+SIGMAGZ*sensorerror;

'apriori estimate of omega1 "measurement" sigma (rad/s): ',estSIGOMEGA1                     %apriori estimate of SIGOMEGA1
'apriori estimate of omega2 "measurement" sigma (rad/s): ',estSIGOMEGA2                     %apriori estimate of SIGOMEGA2
'apriori estimate of omega3 "measurement" sigma (rad/s): ',estSIGOMEGA3                     %apriori estimate of SIGOMEGA3
'apriori estimate of sun sensor x axis measurement sigma (deg): ',estSIGSUNX*180/pi         %apriori estimate of SIGSUNX
'apriori estimate of sun sensor y axis measurement sigma (deg): ',estSIGSUNY*180/pi         %apriori estimate of SIGSUNY
'apriori estimate of sun sensor z axis measurement sigma (deg): ',estSIGSUNZ*180/pi         %apriori estimate of SIGSUNZ
'apriori estimate of mag field sensor x axis measurement sigma (deg): ',estSIGMAGX*180/pi   %apriori estimate of SIGMAGX
'apriori estimate of mag field sensor y axis measurement sigma (deg): ',estSIGMAGY*180/pi   %apriori estimate of SIGMAGY
'apriori estimate of mag field sensor z axis measurement sigma (deg): ',estSIGMAGZ*180/pi   %apriori estimate of SIGMAGZ

else

%...or can manually enter whatever intialization values you wish...

estSIGOMEGA1=input('Please enter sigma for the omega1 derived "measurement" in rpm:  ');  %apriori estimate of SIGOMEGA1
estSIGOMEGA2=input('Please enter sigma for the omega2 derived "measurement" in rpm:  ');  %apriori estimate of SIGOMEGA2
estSIGOMEGA3=input('Please enter sigma for the omega3 derived "measurement" in rpm:  ');  %apriori estimate of SIGOMEGA3

estSIGOMEGA1=estSIGOMEGA1*2*pi/60;                                          %convert estSIGOMEGA1 to rad/s
estSIGOMEGA2=estSIGOMEGA2*2*pi/60;                                          %convert estSIGOMEGA2 to rad/s
estSIGOMEGA3=estSIGOMEGA3*2*pi/60;                                          %convert estSIGOMEGA3 to rad/s

estSIGSUNX=input('Please enter sigma for the sun sensor x axis measurement in deg:  ');     %apriori estimate of SIGSUNX
estSIGSUNY=input('Please enter sigma for the sun sensor y axis measurement in deg:  ');     %apriori estimate of SIGSUNY
estSIGSUNZ=input('Please enter sigma for the sun sensor z axis measurement in deg:  ');     %apriori estimate of SIGSUNZ
```

```
estSIGMAGX=input('Please enter sigma for the mag field sensor x axis measurement in deg: '); %apriori estimate of SIGMAGX
estSIGMAGY=input('Please enter sigma for the mag field sensor y axis measurement in deg: '); %apriori estimate of SIGMAGY
estSIGMAGZ=input('Please enter sigma for the mag field sensor z axis measurement in deg: '); %apriori estimate of SIGMAGZ

estSIGSUNX=estSIGSUNX*pi/180;                %convert from degrees to radians
estSIGSUNY=estSIGSUNY*pi/180;                %convert from degrees to radians
estSIGSUNZ=estSIGSUNZ*pi/180;                %convert from degrees to radians
estSIGMAGX=estSIGMAGX*pi/180;                %convert from degrees to radians
estSIGMAGY=estSIGMAGY*pi/180;                %convert from degrees to radians
estSIGMAGZ=estSIGMAGZ*pi/180;                 %convert from degrees to radians

end

'Please press "return" to continue...'
pause;clc;close;

%***** Non-zero submatrices of Measurement Noise Matrix, R *****

ROMEGA=diag([estSIGOMEGA1^2+eps  estSIGOMEGA2^2+eps   estSIGOMEGA3^2+eps]);
RSENSOR=diag([estSIGSUNX^2+eps  estSIGSUNY^2+eps   estSIGSUNZ^2+eps   estSIGMAGX^2+eps   estSIGMAGY^2+eps
estSIGMAGZ^2+eps]);

%***** user input of process noise to reflect uncertainty in dynamic model, PHIS *****
''
confidence=input('Enter process noise to reflect uncertainty in model (default is 112): ');

if isempty(confidence)
  confidence=112;
end

PHIS=confidence;          %(100-confidence)/100;                  %[17] pg. 317

'Process noise applied is: ',PHIS

'press "return" to continue...'
pause;clc;close;

%***** loop to calculate estimated states at each time step *****

count=1;
                                                          %initialize counter
t=0.;
                                                          %initialize time
ArrayT(1)=t;
```

```
%***** time constants and noise variances for noise driving motion *****
%***** right now, the time constants are hard-coded *****
'enter time constants associated with motion...'
''

tauomega1 = input('enter tau omega1 (default=36)    ');
if isempty(tauomega1)
   tauomega1=36
end
tauomega2 = input('enter tau omega2 (default = tauomega1)    ');
if isempty(tauomega2)
   tauomega2=tauomega1
end
tauomega3 = input('enter tau omega3 (default=1e7)    ');
if isempty(tauomega3)
   tauomega3=1e7
end

clc

while count<=length(omega1meas)-1


  %***** Assign variables to more managable variable names *****

  x1=omega1hat(count);
  x2=omega2hat(count);
  x3=omega3hat(count);
  x4=q1hat(count);
  x5=q2hat(count);
  x6=q3hat(count);
  x7=q4hat(count);

  den=sqrt(x4^2+x5^2+x6^2+x7^2);          %convenient compilation of terms to be used later
  den3=den^3;                             %convenient compilation of terms to be used later
  f4=(x1*x7-x2*x6+x3*x5);                 %convenient compilation of terms to be used later
  f5=(x1*x6+x2*x7-x3*x4);                 %convenient compilation of terms to be used later
  f6=(-x1*x5+x2*x4+x3*x7);                %convenient compilation of terms to be used later
  f7=(-x1*x4-x2*x5-x3*x6);                %convenient compilation of terms to be used later

  %***** Calculate elements of linearized State Transition Matrix, F *****

  F(1,1)=-1/tauomega1;                    %dx1x1
```

```
F(1,2)=0;                              %dx1x2
F(1,3)=0;                              %dx1x3
F(1,4)=0;                              %dx1x4
F(1,5)=0;                              %dx1x5
F(1,6)=0;                              %dx1x6
F(1,7)=0;                              %dx1x7
F(2,1)=0;                              %dx2x1
F(2,2)=-1/tauomega2;                   %dx2x2
F(2,3)=0;                              %dx2x3
F(2,4)=0;                              %dx2x4
F(2,5)=0;                              %dx2x5
F(2,6)=0;                              %dx2x6
F(2,7)=0;                              %dx2x7
F(3,1)=0;                              %dx3x1
F(3,2)=0;                              %dx3x2
F(3,3)=-1/tauomega3;                   %dx3x3
F(3,4)=0;                              %dx3x4
F(3,5)=0;                              %dx3x5
F(3,6)=0;                              %dx3x6
F(3,7)=0;                              %dx3x7
F(4,1)=.5*x7/den;                      %dx4x1
F(4,2)=.5*(-x6)/den;                   %dx4x2
F(4,3)=.5*x5/den;                      %dx4x3
F(4,4)=.5*f4*(-x4)/den3;               %dx4x4
F(4,5)=.5*(x3/den+f4*(-x5)/den3);     %dx4x5
F(4,6)=.5*(-x2/den-f4*x6/den3);        %dx4x6
F(4,7)=.5*(x1/den+f4*(-x7)/den3);     %dx4x7
F(5,1)=.5*x6/den;                      %dx5x1
F(5,2)=.5*x7/den;                      %dx5x2
F(5,3)=.5*(-x4)/den;                   %dx5x3
F(5,4)=.5*(-x3/den+f5*(-x4)/den3);     %dx5x4
F(5,5)=.5*f5*(-x5)/den3;               %dx5x5
F(5,6)=.5*(x1/den+f5*(-x6)/den3);     %dx5x6
F(5,7)=.5*(x2/den+f5*(-x7)/den3);     %dx5x7
F(6,1)=.5*(-x5)/den;                   %dx6x1
F(6,2)=.5*x4/den;                      %dx6x2
F(6,3)=.5*x7/den;                      %dx6x3
F(6,4)=.5*(x2/den+f6*(-x4)/den3);     %dx6x4
F(6,5)=.5*(-x1/den+f6*(-x5)/den3);     %dx6x5
F(6,6)=.5*f6*(-x6)/den3;               %dx6x6
F(6,7)=.5*(x3/den+f6*(-x7)/den3);     %dx6x7
F(7,1)=.5*(-x4)/den;                   %dx7x1
F(7,2)=.5*(-x5)/den;                   %dx7x2
F(7,3)=.5*(-x6)/den;                   %dx7x3
```

```
F(7,4)=.5*(-x1/den+f7*(-x4)/den3);        %dx7x4
F(7,5)=.5*(-x2/den+f7*(-x5)/den3);        %dx7x5
F(7,6)=.5*(-x3/den+f7*(-x6)/den3);        %dx7x6
F(7,7)=.5*f7*(-x7)/den3;                  %dx7x7

%***** Use first two terms of infinite Taylor Series expansion to form non-zero elements of Discrete Fundamental Matrix, PHI
*****

PHI=eye(7)+F*t;

%***** Calculate non-zero elements of Discrete Process Noise Covariance Matrix (old Q)*****

num1=(.5*TS^2-TS^3/(3*tauomega1));        %convenient compilation of terms to be used later
num2=(.5*TS^2-TS^3/(3*tauomega2));        %convenient compilation of terms to be used later
num3=(.5*TS^2-TS^3/(3*tauomega3));        %convenient compilation of terms to be used later

Q(1,1)=estSIGOMEGA1^2*(TS-TS^2/tauomega1+TS^3/(3*tauomega1^2));
Q(1,4)=estSIGOMEGA1^2*F(4,1)*num1;
Q(1,5)=estSIGOMEGA1^2*F(1,5)*num1;
Q(1,6)=estSIGOMEGA1^2*F(1,6)*num1;
Q(1,7)=estSIGOMEGA1^2*F(1,7)*num1;
Q(2,2)=estSIGOMEGA2^2*(TS-TS^2/tauomega2*TS^3/(3*tauomega2^2));
Q(2,4)=estSIGOMEGA2^2*F(4,2)*num2;
Q(2,5)=estSIGOMEGA2^2*F(5,2)*num2;
Q(2,6)=estSIGOMEGA2^2*F(6,2)*num2;
Q(2,7)=estSIGOMEGA2^2*F(7,2)*num2;
Q(3,3)=estSIGOMEGA3^2*(TS-TS^2/tauomega3*TS^3/(3*tauomega3^2));
Q(3,4)=estSIGOMEGA3^2*F(4,3)*num3;
Q(3,5)=estSIGOMEGA3^2*F(5,3)*num3;
Q(3,6)=estSIGOMEGA3^2*F(6,3)*num3;
Q(3,7)=estSIGOMEGA3^2*F(7,3)*num3;
Q(4,1)=Q(1,4);
Q(4,2)=Q(2,4);
Q(4,3)=Q(3,4);
Q(4,4)=estSIGOMEGA1^2*(F(4,1)^2+estSIGOMEGA2^2*F(4,2)^2+estSIGOMEGA3^2*F(4,3)^2)*TS^3/3;
Q(5,1)=Q(1,5);
Q(5,2)=Q(2,5);
Q(5,3)=Q(3,5);
Q(5,4)=Q(4,5);
Q(5,5)=estSIGOMEGA1^2*(F(5,1)^2+estSIGOMEGA2^2*F(5,2)^2+estSIGOMEGA3^2*F(5,3)^2)*TS^3/3;
Q(5,6)=(estSIGOMEGA1^2*F(5,1)*F(6,1)+estSIGOMEGA2^2*F(5,2)*F(6,2)+estSIGOMEGA3^2*F(5,3)*F(6,3))*TS^3/3;
Q(5,7)=(estSIGOMEGA1^2*F(5,1)*F(7,1)+estSIGOMEGA2^2*F(5,2)*F(7,2)+estSIGOMEGA3^2*F(5,3)*F(7,3))*TS^3/3;
Q(6,1)=Q(1,6);
Q(6,2)=Q(2,6);
```

```
Q(6,3)=Q(3,6);
Q(6,4)=Q(4,6);
Q(6,5)=Q(5,6);
Q(6,6)=estSIGOMEGA1^2*(F(6,1)^2+estSIGOMEGA2^2*F(6,2)^2+estSIGOMEGA3^2*F(6,3)^2)*TS^3/3;
Q(6,7)=(estSIGOMEGA1^2*F(6,1)*F(7,1)+estSIGOMEGA2^2*F(6,2)*F(7,2)+estSIGOMEGA3^2*F(6,3)*F(7,3))*TS^3/3;
Q(7,1)=Q(1,7);
Q(7,2)=Q(2,7);
Q(7,3)=Q(3,7);
Q(7,4)=Q(4,7);
Q(7,5)=Q(5,7);
Q(7,6)=Q(6,7);
Q(7,7)=estSIGOMEGA1^2*(F(7,1)^2+estSIGOMEGA2^2*F(7,2)^2+estSIGOMEGA3^2*F(7,3)^2)*TS^3/3;

Q=PHIS*Q;

%***** Calculate covariance before update, M *****

PHIT=PHI';                                   %calculate transpose of PHI;
PHIP=PHI*P;
PHIPPHIT=PHIP*PHIT;

M=PHIPPHIT+Q;                                %calculate covariance before update
                                             %(using old P, old PHI, and old Q)

t=t+TS;                                      %increment time

%***** Propagate estimated states forward using one-step Euler integration *****

S=0;                                         %initialize integration timer

while S<=(TS-.0001)                          %loops from current step to just short of next time step

  x1dot=-1/tauomega1*x1;
  x1=x1+H1*x1dot;

  x2dot=-1/tauomega2*x2;
  x2=x2+H1*x2dot;

  x3dot=-1/tauomega3*x3;
  x3=x3+H1*x3dot;

  x4dot=.5*f4/den;
  x4=x4+H1*x4dot;
```

```
    x5dot=.5*f5/den;
    x5=x5+H1*x5dot;

    x6dot=.5*f6/den;
    x6=x6+H1*x6dot;

    x7dot=.5*f7/den;
    x7=x7+H1*x7dot;

    S=S+H1;                                  %increment integration timer by defined integration step size
end                                          %yields each state estimate integrated forward to next time step

  %***** assign values from numerical integration to arrays *****

  xbar=[x1;x2;x3;x4;x5;x6;x7];

  omega1bar(count+1)=x1;
  omega2bar(count+1)=x2;
  omega3bar(count+1)=x3;
  q1bar(count+1)=x4;
  q2bar(count+1)=x5;
  q3bar(count+1)=x6;
  q4bar(count+1)=x7;

  %***** Calculate non-zero elements of the linearized measurement matrix *****

  HMAT=eye(7);                               %in this implementation, measurements are the states

  %***** Calculate Measurement Noise Matrix, RMAT *****
  %***** Uses A matrix from Gauss_newton to relate 6 x 6 RSENSOR to covariance of quaternions after convergence [11] ******

  RMAT=[ROMEGA, zeros(3,4);zeros(4,3), A(:,:,count)*RSENSOR*A(:,:,count)'];

  %***** Calculate Kalman gain matrix *****

  HT=HMAT';                                  %calculate transpose of H1 (next time step)

  HM=HMAT*M;
  HMHT=HM*HT;
  HMHTR=HMHT+RMAT;
  HMHTRINV=inv(HMHTR);
  MHT=M*HT;

  GAIN=MHT*HMHTRINV;                         %Calculate Kalman gain matrix
```

```
    KH=GAIN*HMAT;
    IKH=eye(statesize)-KH;

%***** Calculate covariance matrix after update, P *****

P=IKH*M;                                       %Calculate covariance after update (new P)

%***** Calculate residuals: actual new measurement - projected new measurement *****

%***** Since the H matrix is the identity matrix in this implementation, the "zbar", *****
%***** or projected measurement, is simply the projected state (zbar=xbar)! *****

omega1res=(omega1meas(count+1)-x1);            %calculate residual from propagated state and measurement
omega2res=(omega2meas(count+1)-x2);            %calculate residual from propagated state and measurement
omega3res=(omega3meas(count+1)-x3);            %calculate residual from propagated state and measurement
q1res=q_min(1,count+1)-x4;                     %calculate residual from propagated state and measurement
q2res=q_min(2,count+1)-x5;                     %calculate residual from propagated state and measurement
q3res=q_min(3,count+1)-x6;                     %calculate residual from propagated state and measurement
q4res=q_min(4,count+1)-x7;                     %calculate residual from propagated state and measurement

residual=[omega1res;omega2res;omega3res;q1res;q2res;q3res;q4res];    %form vector of residuals

%***** Calculate updated state estimates using propagated states and Kalman gains

xhat=xbar+GAIN*residual;

%***** Check for loss of data *****

if q_min(1:4,count)==[0 0 0 1]';
  omega1hat(count+1)=omega1hat(count);
  omega2hat(count+1)=omega2hat(count);
  omega3hat(count+1)=omega3hat(count);
  q1hat(count+1)=q1hat(count);
  q2hat(count+1)=q2hat(count);
  q3hat(count+1)=q3hat(count);
  q4hat(count+1)=q4hat(count);
  flag(count)=1;

else

%*************** Data management *****************

  omega1hat(count+1)=xhat(1);
```

```
   omega2hat(count+1)=xhat(2);
   omega3hat(count+1)=xhat(3);
   q1hat(count+1)=xhat(4);
   q2hat(count+1)=xhat(5);
   q3hat(count+1)=xhat(6);
   q4hat(count+1)=xhat(7);
   flag(count)=0;
end
ArrayT(count+1)=t;

%*************** Error Analysis ******************

SP11=sqrt(P(1,1));                      %Calculate theoretical upper error bound on omega1hat
SP11P=-SP11;                            %Calculate theoretical lower error bound on omega1hat
SP22=sqrt(P(2,2));                      %Calculate theoretical upper error bound on omega2hat
SP22P=-SP22;                            %Calculate theoretical lower error bound on omega2hat
SP33=sqrt(P(3,3));                      %Calculate theoretical upper error bound on omega3hat
SP33P=-SP33;                            %Calculate theoretical lower error bound on omega3hat
SP44=sqrt(P(4,4));                      %Calculate theoretical upper error bound on q1hat
SP44P=-SP44;                            %Calculate theoretical lower error bound on q1hat
SP55=sqrt(P(5,5));                      %Calculate theoretical upper error bound on q2hat
SP55P=-SP55;                            %Calculate theoretical lower error bound on q2hat
SP66=sqrt(P(6,6));                      %Calculate theoretical upper error bound on q3hat
SP66P=-SP66;                            %Calculate theoretical lower error bound on q3hat
SP77=sqrt(P(7,7));                      %Calculate theoretical upper error bound on q4hat
SP77P=-SP77;                            %Calculate theoretical lower error bound on q4hat

ArraySP11(count+1)=SP11;
ArraySP11P(count+1)=SP11P;
ArraySP22(count+1)=SP22;
ArraySP22P(count+1)=SP22P;
ArraySP33(count+1)=SP33;
ArraySP33P(count+1)=SP33P;
ArraySP44(count+1)=SP44;
ArraySP44P(count+1)=SP44P;
ArraySP55(count+1)=SP55;
ArraySP55P(count+1)=SP55P;
ArraySP66(count+1)=SP66;
ArraySP66P(count+1)=SP66P;
ArraySP77(count+1)=SP77;
ArraySP77P(count+1)=SP77P;

count=count+1;                          %increment counter
```

```
end

%****** calculate mean square errors for "instantaneous" and for EKF ******
%****** mean square error excludes first 5 samples to reduce effects of transients *****

s2=length(omega1meas);
s1=5;

omega1meas_mse=mean_square_error(omega1meas(s1:s2),omega1(s1:s2));
omega2meas_mse=mean_square_error(omega2meas(s1:s2),omega2(s1:s2));
omega3meas_mse=mean_square_error(omega3meas(s1:s2),omega3(s1:s2));
q1_q_min_mse=mean_square_error(q_min(1,(s1:s2)),q1(s1:s2));
q2_q_min_mse=mean_square_error(q_min(2,(s1:s2)),q2(s1:s2));
q3_q_min_mse=mean_square_error(q_min(3,(s1:s2)),q3(s1:s2));
q4_q_min_mse=mean_square_error(q_min(4,(s1:s2)),q4(s1:s2));

omega1_ekf_mse=mean_square_error(omega1hat(s1:s2),omega1(s1:s2));
omega2_ekf_mse=mean_square_error(omega2hat(s1:s2),omega2(s1:s2));
omega3_ekf_mse=mean_square_error(omega3hat(s1:s2),omega3(s1:s2));
q1_ekf_mse=mean_square_error(q1hat(s1:s2),q1(s1:s2));
q2_ekf_mse=mean_square_error(q2hat(s1:s2),q2(s1:s2));
q3_ekf_mse=mean_square_error(q3hat(s1:s2),q3(s1:s2));
q4_ekf_mse=mean_square_error(q4hat(s1:s2),q4(s1:s2));

%*************** Plotting and output statements *****************

plots=input('Enter "1" to display plots. Enter "0" for no plots. Default is no plots:  ');
if isempty(plots)
  plots=0;
end

if plots==1

%****** plots of error vs. true and theoretical error bounds *******

plot(ArrayT,(omega1(1:s2)-omega1hat),ArrayT,ArraySP11,'x',ArrayT,ArraySP11P,'x');
grid on;title('error in EKF estimate of omega1');
legend('error in omega1hat','theoretical bound');
xlabel('time (sec)');ylabel('error in omega1hat (rad/s)');
pause;close;clc;

plot(ArrayT,(omega2(1:s2)-omega2hat),ArrayT,ArraySP22,'x',ArrayT,ArraySP22P,'x');
grid on;title('error in EKF estimate of omega2');
legend('error in omega2hat','theoretical bound');
```

```
xlabel('time (sec)');ylabel('error in omega2hat (rad/s)');
pause;close;clc;

plot(ArrayT,(omega3(1:s2)-omega3hat),ArrayT,ArraySP33,'x',ArrayT,ArraySP33P,'x');
grid on;title('error in EKF estimate of omega3');
legend('error in omega3hat','theoretical bound');
xlabel('time (sec)');ylabel('error in omega3hat (rad/s)');
pause;close;clc;

plot(ArrayT,(q1(1:s2)-q1hat),ArrayT,ArraySP44,'x',ArrayT,ArraySP44P,'x');
grid on;title('error in EKF estimate of q1');
legend('error in q1hat','theoretical bound');
xlabel('time (sec)');ylabel('error in q1');
pause;close;clc;

plot(ArrayT,(q2(1:s2)-q2hat),ArrayT,ArraySP55,'x',ArrayT,ArraySP55P,'x');
grid on;title('error in EKF estimate of q2');
legend('error in q2hat','theoretical bound');
xlabel('time (sec)');ylabel('error in q2');
pause;close;clc;

plot(ArrayT,(q3(1:s2)-q3hat),ArrayT,ArraySP66,'x',ArrayT,ArraySP66P,'x');
grid on;title('error in EKF estimate of q3');
legend('error in q3hat','theoretical bound');
xlabel('time (sec)');ylabel('error in q3');
pause;close;clc;

plot(ArrayT,(q4(1:s2)-q4hat),ArrayT,ArraySP77,'x',ArrayT,ArraySP77P,'x');
grid on;title('error in EKF estimate of q4');
legend('error in q4hat','theoretical bound');
xlabel('time (sec)');ylabel('error in q4');
pause;close;clc;

%****** plots to show true vs. quaternion derived alone vs. EKF estimates *******

plot( ArrayT, omega1meas,'gx',ArrayT, omega1hat,'m', ArrayT, omega1(1:s2),'--');grid on;
title('omega1 from sensor, EKF estimate of omega1, "True" omega1' );
xlabel('time (sec)');ylabel('omega1 (rad/s)');
legend('omega1 from rate sensor','omega1 from EKF','true omega1');
pause;clc;close;

plot( ArrayT, omega2meas,'gx',ArrayT, omega2hat,'m', ArrayT, omega2(1:s2),'--','MarkerSize',5);grid on;
title('omega2 from sensor, EKF estimate of omega2, "True" omega2' );
xlabel('time (sec)');ylabel('omega2 (rad/s)');
```

```
legend('omega2 from rate sensor','omega2 from EKF','true omega2');
pause;clc;close;

plot( ArrayT, omega3meas,'gx',ArrayT, omega3hat,'m', ArrayT, omega3(1:s2),'--','MarkerSize',5);grid on;
title('omega3 from sensor, EKF estimate of omega3, "True" omega3' );
xlabel('time (sec)');ylabel('omega3 (rad/s)');
legend('omega3 from rate sensor','omega3 from EKF','true omega3');
pause;clc;close;

end

'Summary of performance of each method...'
' '
'The MSE in omega1 from the rate sensor is (rad/s): ',' ',omega1meas_mse
' '
'The MSE in omega1 for the EKF estimates is (rad/s): ',' ',omega1_ekf_mse
' '
'The MSE in omega2 from the rate sensor is (rad/s): ',' ',omega2meas_mse
' '
'The MSE in omega2 for the EKF estimates is (rad/s): ',' ',omega2_ekf_mse
' '
'Please press "return" to continue...'
pause;clc;close;
'The MSE in omega3 derived from the rate sensor is is (rad/s): ',' ',omega3meas_mse
' '
'The MSE in omega3 for the EKF estimates is (rad/s): ',' ',omega3_ekf_mse
' '
'The MSE in q1 derived using Gauss-Newton is: ',' ',q1_q_min_mse
' '
'The MSE in q1 for the EKF estimates is: ',' ',q1_ekf_mse
' '
'Please press "return" to continue...'
pause;clc;close;

'Summary of performance of each method (cont.) ...'
' '
'The MSE in q2 derived using Gauss-Newton is: ',' ',q2_q_min_mse
' '
'The MSE in q2 for the EKF estimates is: ',' ',q2_ekf_mse
' '
'press "return" to continue...'
pause;clc;close;

'Summary of performance of each method (cont.) ...'
```

```
' '
'The MSE in q3 derived using Gauss-Newton is: ',' ',q3_q_min_mse
' '
'The MSE in q3 for the EKF estimates is: ',' ',q3_ekf_mse
' '
'The MSE in q4 derived using Gauss-Newton is: ',' ',q4_q_min_mse
' '
'The MSE in q4 for the EKF estimates is: ',' ',q4_ekf_mse
' '
'press "return" to continue...'
pause;clc;close;

'Summary of performance of each method (cont.) ...'

'****** Evaluate Mean-Square-Error of rotated vectors using Gauss-Newton derived quaternion ******'
'Press "return" to rotate the reference vector at each time step to the'
'representation in the rotated frame using both the actual rotation quaternion'
'and the quaternion derived using the ekf...once rotation is complete,'
'calculate the mean-square-error between the results.'
pause;close;clc;

qekf=[q1hat;q2hat;q3hat;q4hat];                          %matrix representation of ekf quaternions

ekf_mse_error=[ ];
for j=1:count;
   ekfvec(1:6,j)=qframerot6(qekf(1:4,j),vref(1:6,j));     %rotate ref vector using min error quaternion
   Q_vec(1:6,j)=qframerot6(Qvec(1:4,j),vref(1:6,j));      %rotate ref vector using true quaternion
   ekf_mse_error(j)=mean_square_error(ekfvec(1:6,j),Q_vec(1:6,j)); %calculate MSE between rotated vectors
end

for k=1:count
   oekfsunangle(k)=acos(dot(ekfvec(1:3,k),Q_vec(1:3,k))/(norm(ekfvec(1:3,k))*norm(Q_vec(1:3,k))));
   oekfmagangle(k)=acos(dot(ekfvec(4:6,k),Q_vec(4:6,k))/(norm(ekfvec(4:6,k))*norm(Q_vec(4:6,k))));
   oGNsunangle(k)=acos(dot(q_minvec(1:3,k),Q_vec(1:3,k))/(norm(q_minvec(1:3,k))*norm(Q_vec(1:3,k))));
   oGNmagangle(k)=acos(dot(q_minvec(4:6,k),Q_vec(4:6,k))/(norm(q_minvec(4:6,k))*norm(Q_vec(4:6,k))));
end
oekfsunangle=oekfsunangle*180/pi;
oekfmagangle=oekfmagangle*180/pi;
oGNsunangle=oGNsunangle*180/pi;
oGNmagangle=oGNmagangle*180/pi;

%****** Plot vectors rotated using derived and actual quaternions *****
%plot3(Q_vec(1,1:count),Q_vec(2,1:count),Q_vec(3,1:count),'b');grid on;hold on;
%plot3(ekfvec(1,1:count),ekfvec(2,1:count),ekfvec(3,1:count),'m');
```

```
%plot3(Q_vec(4,1:count),Q_vec(5,1:count),Q_vec(6,1:count),'r');grid on;hold on;
%plot3(ekfvec(4,1:count),ekfvec(5,1:count),ekfvec(6,1:count),'g');
%xlabel('X component');ylabel('Y component');zlabel('Z component');
%legend('sun vector, actual q','sun vector, ekf q','mag field vector, actual q', 'mag field vector, ekf q');
%title('3-D plot of reference vectors rotated using actual quaternions and ekf derived quaternions');
%pause;close;clc;

%****** Plot mean-square-error between reference vector rotated using actual quaternion ******
%****** and reference vector rotated using derived quaternion.                    ******

%plot(T(1:count),ekf_mse_error(1:count)),grid on;
%title('mean-square error between reference vectors rotated using ekf derived and actual quaternions');
%xlabel('time (sec)');ylabel('MSE');legend('MSE')

%'press "return" to continue...'
%pause;close;clc;

'********************* Overall Performance Indicator ************************'
'****** Excludes first five measurements to reduce effect of transients ******'
' '
'The average of the omega mean-square-errors from the rate sensors is:  '
GNomegaperf=(omega1meas_mse+omega2meas_mse+omega3meas_mse)/3
' '
'The average of the omega mean-square-errors for the ekf results is:  '
ekfomegaperf=(omega1_ekf_mse+omega2_ekf_mse+omega3_ekf_mse)/3
' '
'The average of the quaternion component mean-square-errors for the Gauss-Newton method is:  '
GNqperf=(q1_q_min_mse+q2_q_min_mse+q3_q_min_mse+q4_q_min_mse)/4
' '
'The average of the quaternion component mean-square-errors for the ekf results is:  '
ekfqperf=(q1_ekf_mse+q2_ekf_mse+q3_ekf_mse+q4_ekf_mse)/4
' '
'press "return" to continue...'
pause;clc;close;

'Summary of performance of each method (cont.) ...'
' '
'The average MSE for the ref vectors rotated using GN quaternions:  '
mean_GN_mse_error=mean(GN_mse_error)
' '
'The average MSE for the ref vectors rotated using ekf quaternions:  '
mean_ekf_mse_error=mean(ekf_mse_error)
' '
'press "return" to continue...'
```

```
pause;clc;close;

'Summary of performance of each method (cont.) ...'
' '
'overall GN performance: sqrt[(mean omega MSE)^2 + (mean rotated vector MSE)^2)]'
overall_GN_perf=sqrt(GNomegaperf^2+mean_GN_mse_error^2)
' '
'overall ekf performance: sqrt[(mean omega MSE)^2 + (mean rotated vector MSE)^2)]'
overall_ekf_perf=sqrt(ekfomegaperf^2+mean_ekf_mse_error^2)
' '
%'**************************************************************************************'
%'********************************* ekf summary **************************************'
%'**************************************************************************************'
'omega:       ',ekfomegaperf
'quaternion:  ',mean_ekf_mse_error
'overall:     ',overall_ekf_perf
'mean sun angle error:    ',mean(oekfsunangle)
'std sun angle error:    ',std(oekfsunangle)
'max sun angle error:    ',max(oekfsunangle)
'mean mag angle error:    ',mean(oekfmagangle)
'std mag angle error:    ',std(oekfmagangle)
'max mag angle error:    ',max(oekfmagangle)
' '
'press "return" to continue...'
pause;close;clc;

angles=input('Enter "1" to calculate attitude error angles. Enter "0" for no angles. Default is no angles:  ');
if isempty(angles)
  angles=0;
end

if angles==1

   %Calculate angles between body frame rotated using estimated quaternion and
   %rotated using "true" quaternion

   [ovec1,ovec2,ovec3,ovec1hat,ovec2hat,ovec3hat,ang1,ang2,ang3]=orientation3(Qvec, qekf);

end



%end of program
```

A.16 Rates_ukf.m

Rates_ukf.m is the MATLAB script that implements the unscented Kalman filter (UKF) designed in Chapter 9. This implementation is for the case where rotational rates are directly measured by rate sensors and the noisy measurements are made available to the filter. Rates_ukf.m is called by attitude_filter.m if the appropriate option is selected by the user.

```
%RATES_UKF - Discrete Unscented Kalman Filter Implementation
%Uses data generated by attitude_sim.m to estimate 3-D body rates and
%the attitude quaternion of a sounding rocket given noisy measurements
%from two attitude sensors and from rate sensors. Uses a Gauss-Newton
%optimization and an Unscented Kalman Filter (UKF) to generate a best estimate.
%
%Written by Mark Charlton. Last modified 9 November 2003.

%Object subject to rotational motion -- non-linear Equation of Motion

%***** Initialize parameters and matrices *****

clc

ics=[ ];
sigmas=[ ];
error=[ ];                                      %initialize error multiplier
sensorerror=[ ];                                %initialize error multiplier
ArrayT=[ ];                                     %initialize Arrays
q1hat=[ ];
q2hat=[ ];
q3hat=[ ];
q4hat=[ ];
omega1hat=[ ];
omega2hat=[ ];
omega3hat=[ ];
ArraySP11=[ ];
ArraySP11P=[ ];
ArraySP22=[ ];
ArraySP22P=[ ];
ArraySP33=[ ];
ArraySP33P=[ ];
ArraySP44=[ ];
ArraySP44P=[ ];
ArraySP55=[ ];
```

```
ArraySP55P=[ ];
ArraySP66=[ ];
ArraySP66P=[ ];
ArraySP77=[ ];
ArraySP77P=[ ];
qukf=[ ];
ukfvec=[ ];
statesize=7;                                    %system order
PHIS=0;                                         %process noise (measure of uncertainty in dynamic model)
Q=zeros(statesize,statesize);                   %initialize process noise matrix
P=zeros(statesize,statesize);                   %initialize covariance matrix
HMAT=zeros(statesize,statesize);                %initialize linearized measurement matrix

%***** Assign initial estimates for omega1hat, omega2hat, omega3hat, q1hat, q2hat, q3hat, q4hat *****

'Do you wish to enter initial estimates of the rotational body rates and the quaternion components'
'or let the program use the actual initial conditions with a user defined percent error to'
'make an initial estimate (default is to let the program do it)?'
' '
'If you choose manual entry, an uncertainty in the estimate is also required. The default is "auto".'
' '
ics=input('Enter "1" for manual entry and "0" for automatic:  ');
if isempty(ics)
  ics=0;
end

if ics == 0

%***** base initialization on actual intial conditions with user defined standard error *****
clc
error=input('Enter percentage error (+/-) in initial conditions (i.e., -5.2% = "-5.2") Default is 5%:  ');
if isempty(error)
  error=5;
end

'percent error applied to the initial conditions is: ',error

error=error/100;

omega1hat(1)=omega1(1)+omega1(1)*error;

omega2hat(1)=omega2(1)+omega2(1)*error;
omega3hat(1)=omega3(1)+omega3(1)*error;
q1hat(1)=q1(1)+q1(1)*error;
```

```
q2hat(1)=q2(1)+q2(1)*error;
q3hat(1)=q3(1)+q3(1)*error;
q4hat(1)=q4(1)+q4(1)*error;

'apriori estimate of omega1 (rad/s): ',omega1hat(1)            %apriori estimate of omega1
'apriori estimate of omega2 (rad/s): ',omega2hat(1)            %apriori estimate of omega2
'apriori estimate of omega3 (rad/s): ',omega3hat(1)            %apriori estimate of omega3
'apriori estimate of q1: ',q1hat(1)                            %apriori estimate of q1
'apriori estimate of q2: ',q2hat(1)                            %apriori estimate of q2
'apriori estimate of q3: ',q3hat(1)                            %apriori estimate of q3
'apriori estimate of q4: ',q4hat(1)                            %apriori estimate of q4

%***** Non-zero entries of initial covariance matrix for auto initial cond *****

P(1,1)=(omega1(1)*error+eps)^2;                                %set at square of uncertainty in omega1
P(2,2)=(omega2(1)*error+eps)^2;                                %set at square of uncertainty in omega2
P(3,3)=(omega3(1)*error+eps)^2;                                %set at square of uncertainty in omega3
P(4,4)=(q1(1)*error+eps)^2;                                    %set at square of uncertainty in q1
P(5,5)=(q2(1)*error+eps)^2;                                    %set at square of uncertainty in q2
P(6,6)=(q3(1)*error+eps)^2;                                    %set at square of uncertainty in q3
P(7,7)=(q4(1)*error+eps)^2;                                    %set at square of uncertainty in q4

else

%...or can manually enter whatever intialization values you wish...

omega1hat(1)=input('Please enter initial omega1 in rpm:  ');   %apriori estimate of omega1
omega2hat(1)=input('Please enter initial omega2 in rpm:  ');   %apriori estimate of omega2
omega3hat(1)=input('Please enter initial omega3 in rpm:  ');   %apriori estimate of omega3
q1hat(1)=input('Please enter initial q1:  ');                  %apriori estimate of q1
q2hat(1)=input('Please enter initial q2:  ');                  %apriori estimate of q2
q3hat(1)=input('Please enter initial q3:  ');                  %apriori estimate of q3
q4hat(1)=input('Please enter initial q4:  ');                  %apriori estimate of q4

omega1hat(1)=omega1hat(1)*2*pi/60;
omega2hat(1)=omega2hat(1)*2*pi/60;
omega3hat(1)=omega3hat(1)*2*pi/60;                             %convert manual omega entries to rad/s

'For manual entry of initial conditions, estimated uncertainty in the initial conditions'
'is necessary to intialize the covariance matrix...'
' '
domega1hat=input('Please enter estimated uncertainty in initial omega1 (rad/s):  ');      %uncertainty in initial estimate of omega1
domega2hat=input('Please enter estimated uncertainty in initial omega2 (rad/s):  ');      %uncertainty in initial estimate of omega2
domega3hat=input('Please enter estimated uncertainty in initial omega3 (rad/s):  ');      %uncertainty in initial estimate of omega3
```

```
dq1hat=input('Please enter estimated uncertainty in initial q1:  ');          %uncertainty in initial estimate of q1
dq2hat=input('Please enter estimated uncertainty in initial q2:  ');          %uncertainty in initial estimate of q2
dq3hat=input('Please enter estimated uncertainty in initial q3:  ');          %uncertainty in initial estimate of q3
dq4hat=input('Please enter estimated uncertainty in initial q4:  ');          %uncertainty in initial estimate of q4

%***** Non-zero entries of initial covariance matrix for manual initial cond *****

P(1,1)=(domega1hat+eps)^2;                   %set at square of uncertainty in initial estimate of omega1
P(2,2)=(domega2hat+eps)^2;                   %set at square of uncertainty in initial estimate of omega2
P(3,3)=(domega3hat+eps)^2;                   %set at square of uncertainty in initial estimate of omega3
P(4,4)=(dq1hat+eps)^2;                       %set at square of uncertainty in initial estimate of q1
P(5,5)=(dq2hat+eps)^2;                       %set at square of uncertainty in initial estimate of q2
P(6,6)=(dq3hat+eps)^2;                       %set at square of uncertainty in initial estimate of q3
P(7,7)=(dq4hat+eps)^2;                       %set at square of uncertainty in initial estimate of q4

end

%insure initial quaternion estimate is a unit quaternion

mag=sqrt(q1hat(1)^2+q2hat(1)^2+q3hat(1)^2+q4hat(1)^2);
q1hat(1)=q1hat(1)/mag;
q2hat(1)=q2hat(1)/mag;
q3hat(1)=q3hat(1)/mag;
q4hat(1)=q4hat(1)/mag;

%***** Enter measurement noise values for use in calculating Measurement Noise Matrix, R *****

'Do you wish to enter sensor measurement sigmas manually or let the program use the actual'
'sensor measurement sigmas with a user defined percent error to make an initial estimate'
'(default is to let the program do it)?'
''
sigmas=input('Enter "1" for manual entry and "0" for automatic:  ');
if isempty(sigmas)
  sigmas=0;
end
clc
if sigmas == 0

%***** base sensor noise matrix on actual sigmas plus user defined error *****

sensorerror=input('Enter percentage error (+/-) in sensor measurement sigmas (i.e., -5.2% = "-5.2") Default is 0%:  ');
if isempty(sensorerror)
  sensorerror=0;
end
```

```
'percent error applied to the actual sensor measurement sigmas is: ',sensorerror

sensorerror=sensorerror/100;

SIGOMEGA1=stdomega1meas;           %right now...calculated in attitude_filter from std of actual error
SIGOMEGA2=stdomega2meas;           %right now...calculated in attitude_filter from std of actual error
SIGOMEGA3=stdomega3meas;           %right now...calculated in attitude_filter from std of actual error
estSIGOMEGA1=SIGOMEGA1+SIGOMEGA1*sensorerror;
estSIGOMEGA2=SIGOMEGA2+SIGOMEGA2*sensorerror;
estSIGOMEGA3=SIGOMEGA3+SIGOMEGA3*sensorerror;
estSIGSUNX=SIGSUNX+SIGSUNX*sensorerror;
estSIGSUNY=SIGSUNY+SIGSUNY*sensorerror;
estSIGSUNZ=SIGSUNZ+SIGSUNZ*sensorerror;
estSIGMAGX=SIGMAGX+SIGMAGX*sensorerror;
estSIGMAGY=SIGMAGY+SIGMAGY*sensorerror;
estSIGMAGZ=SIGMAGZ+SIGMAGZ*sensorerror;

'apriori estimate of omega1 "measurement" sigma (rad/s): ',estSIGOMEGA1                    %apriori estimate of SIGOMEGA1
'apriori estimate of omega2 "measurement" sigma (rad/s): ',estSIGOMEGA2                    %apriori estimate of SIGOMEGA2
'apriori estimate of omega3 "measurement" sigma (rad/s): ',estSIGOMEGA3                    %apriori estimate of SIGOMEGA3
'apriori estimate of sun sensor x axis measurement sigma (deg): ',estSIGSUNX*180/pi        %apriori estimate of SIGSUNX
'apriori estimate of sun sensor y axis measurement sigma (deg): ',estSIGSUNY*180/pi        %apriori estimate of SIGSUNY
'apriori estimate of sun sensor z axis measurement sigma (deg): ',estSIGSUNZ*180/pi        %apriori estimate of SIGSUNZ
'apriori estimate of mag field sensor x axis measurement sigma (deg): ',estSIGMAGX*180/pi  %apriori estimate of SIGMAGX
'apriori estimate of mag field sensor y axis measurement sigma (deg): ',estSIGMAGY*180/pi  %apriori estimate of SIGMAGY
'apriori estimate of mag field sensor z axis measurement sigma (deg): ',estSIGMAGZ*180/pi  %apriori estimate of SIGMAGZ

else

%...or can manually enter whatever intialization values you wish...

estSIGOMEGA1=input('Please enter sigma for the omega1 derived "measurement" in rpm:  '); %apriori estimate of SIGOMEGA1
estSIGOMEGA2=input('Please enter sigma for the omega2 derived "measurement" in rpm:  '); %apriori estimate of SIGOMEGA2
estSIGOMEGA3=input('Please enter sigma for the omega3 derived "measurement" in rpm:  '); %apriori estimate of SIGOMEGA3

estSIGOMEGA1=estSIGOMEGA1*2*pi/60;                                                        %convert estSIGOMEGA1 to rad/s
estSIGOMEGA2=estSIGOMEGA2*2*pi/60;                                                        %convert estSIGOMEGA2 to rad/s
estSIGOMEGA3=estSIGOMEGA3*2*pi/60;                                                        %convert estSIGOMEGA3 to rad/s

estSIGSUNX=input('Please enter sigma for the sun sensor x axis measurement in deg:  ');      %apriori estimate of SIGSUNX
estSIGSUNY=input('Please enter sigma for the sun sensor y axis measurement in deg:  ');      %apriori estimate of SIGSUNY
estSIGSUNZ=input('Please enter sigma for the sun sensor z axis measurement in deg:  ');      %apriori estimate of SIGSUNZ
estSIGMAGX=input('Please enter sigma for the mag field sensor x axis measurement in deg:  ');%apriori estimate of SIGMAGX
```

```
estSIGMAGY=input('Please enter sigma for the mag field sensor y axis measurement in deg:  ');%apriori estimate of SIGMAGY
estSIGMAGZ=input('Please enter sigma for the mag field sensor z axis measurement in deg:  ');%apriori estimate of SIGMAGZ

estSIGSUNX=estSIGSUNX*pi/180;                          %convert from degrees to radians
estSIGSUNY=estSIGSUNY*pi/180;                          %convert from degrees to radians
estSIGSUNZ=estSIGSUNZ*pi/180;                          %convert from degrees to radians
estSIGMAGX=estSIGMAGX*pi/180;                          %convert from degrees to radians
estSIGMAGY=estSIGMAGY*pi/180;                          %convert from degrees to radians
estSIGMAGZ=estSIGMAGZ*pi/180;                          %convert from degrees to radians

end

'Please press "return" to continue...'
pause;clc;close;

%***** Non-zero submatrices of Measurement Noise Matrix, R  *****

ROMEGA=diag([estSIGOMEGA1^2+eps  estSIGOMEGA2^2+eps   estSIGOMEGA3^2+eps]);
RSENSOR=diag([estSIGSUNX^2+eps estSIGSUNY^2+eps   estSIGSUNZ^2+eps   estSIGMAGX^2+eps   estSIGMAGY^2+eps
estSIGMAGZ^2+eps]);

%***** user input of process noise to reflect uncertainty in dynamic model, PHIS *****
' '
confidence=input('Enter process noise to reflect uncertainty in model (default is .023):  ');

if isempty(confidence)
  confidence=.023;
end

PHIS=confidence;          %(100-confidence)/100;  %[17] pg. 317

'Process noise applied is: ',PHIS

for i=1:1:statesize;
  Q(i,i)=PHIS^2;
end
' '
%***** time constants and noise variances for noise driving motion *****

'enter time constants associated with motion...'
' '

tauomega1 = input('enter tau omega1 (default=.34)     ');
if isempty(tauomega1)
```

```
   tauomega1=.34
end
tauomega2 = input('enter tau omega2 (default = .34)    ');
if isempty(tauomega2)
   tauomega2=.34
end
tauomega3 = input('enter tau omega3 (default=1e7)    ');
if isempty(tauomega3)
   tauomega3=1e7
end

'press "return" to continue...'
pause;clc;close;

'Enter the scaling parameters for the Unscented Transform...'
' '
alphd=input('enter alpha (default is .01):  ');                 %determines spread of sigma values about mean, .0001<alpha<1
if isempty (alpha)
  alpha=.01
end
' '
beta=input('enter beta (default is 2):  ');                     %incorporates prior knowledge of state dist, beta=2 for Gaussian
if isempty (beta)
  beta=2
end
' '
augalpha=input('enter alpha for augmented sigma (default is .01):  ');
if isempty (augalpha)
  augalpha=.01
end
                                                                %determines spread of sigma values about mean, .0001<alpha<1
' '
augbeta=input('enter beta for augmented sigma (default is 2):  ');
if isempty (augbeta)
  augbeta=2
end
                                                                %incorporates prior knowledge of state dist, beta=2 for Gaussian

' '
'Please wait while data is processed...'

%***** loop to calculate estimated states at each time step *****
```

```
count=1;
                                                    %initialize counter
t=0.;
                                                    %initialize time
ArrayT(1)=t;

while count<=length(omega1meas)-1

  %***** Assign variables to more managable variable names *****

  x1=omega1hat(count);
  x2=omega2hat(count);
  x3=omega3hat(count);
  x4=q1hat(count);
  x5=q2hat(count);
  x6=q3hat(count);
  x7=q4hat(count);

  den=sqrt(x4^2+x5^2+x6^2+x7^2);                    %convenient compilation of terms to be used later
  den3=den^3;                                       %convenient compilation of terms to be used later
  f4=(x1*x7-x2*x6+x3*x5);                           %convenient compilation of terms to be used later
  f5=(x1*x6+x2*x7-x3*x4);                           %convenient compilation of terms to be used later
  f6=(-x1*x5+x2*x4+x3*x7);                          %convenient compilation of terms to be used later
  f7=(-x1*x4-x2*x5-x3*x6);                          %convenient compilation of terms to be used later

  %***** Current estimate of the state vector *****

  eststate=[x1 x2 x3 x4 x5 x6 x7]';

  %***** Calculate parameter values *****

  L=statesize;                                      %dimension of state vector

  kappa=3-L;                                        %secondary scaling parameter
  lambda=alpha^2*(L+kappa)-L;                       %primary scaling parameter
  gamma=sqrt(L+lambda);

  if isempty(find(eig(P)<0))                        %check for positive-definiteness of P
     C=chol(P);
  else
    C=zeros(statesize,statesize);                   %Cholesky factorization of covariance matrix
  end

  %***** Calculate Sigma Point Matrix, sigma *****
```

```
sigma=[ ];
sigma(:,1)=eststate;                            %Sigma-zero
for i=2:1:L+1
  sigma(:,i)=sigma(:,1)+gamma*C(i-1,:)';        %Sigma points 1 through L are eststate plus gamma times the
                                                %transpose of the ith row of the Cholesky upper triangular
                                                %factorization
end

for i=L+2:1:2*L+1
  sigma(:,i)=sigma(:,1)-gamma*C(i-(L+1),:)';    %Sigma points L+1 through 2L are eststate minus gamma
                                                %times the transpose of the ith row of the Cholesky upper
                                                %triangular factorization
end

%***** Instantiate Sigma Points Through Nonlinear Function, sigmabar *****
%***** Propagate estimated states forward using one-step Euler integration *****

sigmabar=[ ];

for j=1:1:length(sigma(1,:))

   X1=sigma(1,j);                               %omega1hat component to be propagated forward one time step
   X2=sigma(2,j);                               %omega2hat component to be propagated forward one time step
   X3=sigma(3,j);                               %omega3hat component to be propagated forward one time step
   X4=sigma(4,j);                               %q1hat component to be propagated forward one time step
   X5=sigma(5,j);                               %q2hat component to be propagated forward one time step
   X6=sigma(6,j);                               %q3hat component to be propagated forward one time step
   X7=sigma(7,j);                               %q4hat component to be propagated forward one time step

S=0;                                            %initialize integration timer
H1=.01;                                         %set step size for numerical integration

while S<=(TS-.0001)                             %loops from current step to just short of next time step

   X1dot=-1/tauomega1*X1;
   X1=X1+H1*X1dot;

   X2dot=-1/tauomega2*X2;
   X2=X2+H1*X2dot;

   X3dot=-1/tauomega3*X3;
   X3=X3+H1*X3dot;
```

```
    X4dot=.5*f4/den;
    X4=X4+H1*X4dot;

    X5dot=.5*f5/den;
    X5=X5+H1*X5dot;

    X6dot=.5*f6/den;
    X6=X6+H1*X6dot;

    X7dot=.5*f7/den;
    X7=X7+H1*X7dot;

    S=S+H1;                                   %increment integration timer by defined integration step size
end                                           %yields each state estimate integrated forward to next time step

  sigmabar(1,j)=X1;                           %transformed sigma points
  sigmabar(2,j)=X2;
  sigmabar(3,j)=X3;
  sigmabar(4,j)=X4;
  sigmabar(5,j)=X5;
  sigmabar(6,j)=X6;
  sigmabar(7,j)=X7;

  end

  %***** Compute the predicted mean of each state *****

  Wm0=lambda/(L+lambda);                      %weight Wm for initial step
  Wm=1/(2*(L+lambda));                        %Wm for steps other than zero

  X1bar=Wm0*sigmabar(1,1);
  X2bar=Wm0*sigmabar(2,1);
  X3bar=Wm0*sigmabar(3,1);
  X4bar=Wm0*sigmabar(4,1);                    %this block predicts mean as a weighted sum
  X5bar=Wm0*sigmabar(5,1);
  X6bar=Wm0*sigmabar(6,1);
  X7bar=Wm0*sigmabar(7,1);

  for k=2:1:2*L+1
    X1bar=X1bar+Wm*sigmabar(1,k);
    X2bar=X2bar+Wm*sigmabar(2,k);
    X3bar=X3bar+Wm*sigmabar(3,k);
    X4bar=X4bar+Wm*sigmabar(4,k);
    X5bar=X5bar+Wm*sigmabar(5,k);
    X6bar=X6bar+Wm*sigmabar(6,k);
    X7bar=X7bar+Wm*sigmabar(7,k);
  end

  statebar=[X1bar X2bar X3bar X4bar X5bar X6bar X7bar]';         %vector of projected states

  %***** Compute the predicted covariance before the update, M *****

  Wc0=lambda/(L+lambda)+1-alpha^2+beta;                          %weight Wc for initial step
  Wc=1/(2*(L+lambda));                                           %Wc for steps other than zero

  M=Wc0*((sigmabar(:,1)-statebar)*(sigmabar(:,1)-statebar)');

  for m=2:1:2*L+1
    M=M+Wc*((sigmabar(:,m)-statebar)*(sigmabar(:,m)-statebar)');
  end

  %***** Augment Sigma point Matrix to Account for Process Noise Covariance, Q *****

  Lprime=2*L;                                                    %augment with additional points
  augkappa=3-Lprime;                                             %secondary scaling parameter
  auglambda=augalpha^2*(Lprime+augkappa)-Lprime;                 %primary scaling parameter
  auggamma=sqrt(Lprime+auglambda);                               %gamma for augmented matrix
  if isempty(find(eig(Q)<0))                                     %check for positive-definiteness
    C2=chol(Q);
  else
    C2=zeros(statesize,statesize);                               %Cholesky factorization of process noise covariance matrix
  end

  newsigma=sigmabar;
  for n=1:1:statesize
    newsigma(:,2*L+1+n)=sigmabar(:,1)+auggamma*C2(n,:)';
  end

  for n=statesize+1:1:2*statesize
    newsigma(:,2*L+1+n)=sigmabar(:,1)-auggamma*C2(n-L,:)';
  end

%***** Instantiate transformed Sigma points through non-linear observation model *****
  %***** In this implementation, the observations are the states (zetabar=newsigma)!*****

  HMAT=eye(7);                                                   %in this implementation, measurements are the states

  for p=1:1:length(newsigma(1,:))
```

```
   zetabar(1,p)=newsigma(1,p);
   zetabar(2,p)=newsigma(2,p);
   zetabar(3,p)=newsigma(3,p);
   zetabar(4,p)=newsigma(4,p);
   zetabar(5,p)=newsigma(5,p);
   zetabar(6,p)=newsigma(6,p);
   zetabar(7,p)=newsigma(7,p);
end

%***** Compute the predicted observation vector, obsbar *****

augWm0=auglambda/(Lprime+auglambda);                 %weight Wm for initial step
augWm=1/(2*(Lprime+auglambda));                      %Wm for steps other than zero

z1bar=augWm0*zetabar(1,1);
z2bar=augWm0*zetabar(2,1);
z3bar=augWm0*zetabar(3,1);
z4bar=augWm0*zetabar(4,1);                           %this block predicts mean as a weighted sum
z5bar=augWm0*zetabar(5,1);
z6bar=augWm0*zetabar(6,1);
z7bar=augWm0*zetabar(7,1);

for k=2:1:length(newsigma(1,:))
   z1bar=z1bar+augWm*zetabar(1,k);
   z2bar=z2bar+augWm*zetabar(2,k);
   z3bar=z3bar+augWm*zetabar(3,k);
   z4bar=z4bar+augWm*zetabar(4,k);
   z5bar=z5bar+augWm*zetabar(5,k);
   z6bar=z6bar+augWm*zetabar(6,k);
   z7bar=z7bar+augWm*zetabar(7,k);
end

obsbar=[z1bar z2bar z3bar z4bar z5bar z6bar z7bar]';   %vector of projected observations

%***** Calculate Measurement Noise Matrix, RMAT *****
%***** Uses A matrix from Gauss_newton to relate 6 x 6 RSENSOR to covariance of quaternions after convergence [11] ******

RMAT=[ROMEGA, zeros(3,4);zeros(4,3), A(:,:,count)*RSENSOR*A(:,:,count)'];

%***** Calculate Innovation Covariance Matrix, Pzz *****

augWc0=auglambda/(Lprime+auglambda)+1-augalpha^2+augbeta;     %weight Wc for initial step
augWc=1/(2*(Lprime+auglambda));                               %Wc for steps other than zero
Pzz=augWc0*((zetabar(:,1)-obsbar)*(zetabar(:,1)-obsbar)');
```

```
for r=2:1:2*Lprime+1
   Pzz=Pzz+augWc*((zetabar(:,r)-obsbar)*(zetabar(:,r)-obsbar)');
end

Pzz=Pzz+RMAT;

%***** Calculate Cross-Correlation Matrix, Pxz *****

Pxz=augWc0*((newsigma(:,1)-statebar)*(zetabar(:,1)-obsbar)');

for r=2:1:2*Lprime+1
   Pxz=Pxz+augWc*((newsigma(:,r)-statebar)*(zetabar(:,r)-obsbar)');
end

%***** Calculate Kalman Gain Matrix *****

K=Pxz*inv(Pzz);                                          %calculate Kalman gain matrix

%***** Form observation vector (measurement) *****

obs(1:7,count+1)=[omega1meas(count+1) omega2meas(count+1) omega3meas(count+1) q_min(1,count+1) q_min(2,count+1)
q_min(3,count+1) q_min(4,count+1)]';

%***** Calculate updated state estimates using propagated states, residuals and Kalman gains

residual=(obsbar-obs(1:7,count+1));
statehat=statebar-K*residual;

%***** Calculate Covariance After the Update *****

P=M-K*Pzz*K';                                            %calcuate covariance after update
P=sqrt(P.^2);

%***** Increment time *****

t=t+TS;                                                  %increment time

%***** Check for loss of data *****

if q_min(1:4,count)==[0 0 0 1]';

   omega1hat(count+1)=omega1hat(count);
   omega2hat(count+1)=omega2hat(count);
   omega3hat(count+1)=omega3hat(count);
```

```
    q1hat(count+1)=q1hat(count);
    q2hat(count+1)=q2hat(count);
    q3hat(count+1)=q3hat(count);
    q4hat(count+1)=q4hat(count);
    flag(count)=1;

else

    %*************** Data management *****************
    omega1hat(count+1)=statehat(1);
    omega2hat(count+1)=statehat(2);
    omega3hat(count+1)=statehat(3);
    q1hat(count+1)=statehat(4);
    q2hat(count+1)=statehat(5);
    q3hat(count+1)=statehat(6);
    q4hat(count+1)=statehat(7);
end

%************** Calculate Covariance of the Residual ****************

covRES=HMAT*M*HMAT'+RMAT;

ArrayT(count+1)=t;

%*************** Error Analysis ******************

SP11=sqrt(P(1,1));          %Calculate theoretical upper error bound on omega1hat
SP11P=-SP11;                %Calculate theoretical lower error bound on omega1hat
SP22=sqrt(P(2,2));          %Calculate theoretical upper error bound on omega2hat
SP22P=-SP22;                %Calculate theoretical lower error bound on omega2hat
SP33=sqrt(P(3,3));          %Calculate theoretical upper error bound on omega3hat
SP33P=-SP33;                %Calculate theoretical lower error bound on omega3hat
SP44=sqrt(P(4,4));          %Calculate theoretical upper error bound on q1hat
SP44P=-SP44;                %Calculate theoretical lower error bound on q1hat
SP55=sqrt(P(5,5));          %Calculate theoretical upper error bound on q2hat
SP55P=-SP55;                %Calculate theoretical lower error bound on q2hat
SP66=sqrt(P(6,6));          %Calculate theoretical upper error bound on q3hat
SP66P=-SP66;                %Calculate theoretical lower error bound on q3hat
SP77=sqrt(P(7,7));          %Calculate theoretical upper error bound on q4hat
SP77P=-SP77;                %Calculate theoretical lower error bound on q4hat

ArraySP11(count+1)=SP11;
ArraySP11P(count+1)=SP11P;
ArraySP22(count+1)=SP22;
```

```
ArraySP22P(count+1)=SP22P;
ArraySP33(count+1)=SP33;
ArraySP33P(count+1)=SP33P;
ArraySP44(count+1)=SP44;
ArraySP44P(count+1)=SP44P;
ArraySP55(count+1)=SP55;
ArraySP55P(count+1)=SP55P;
ArraySP66(count+1)=SP66;
ArraySP66P(count+1)=SP66P;
ArraySP77(count+1)=SP77;
ArraySP77P(count+1)=SP77P;

%************* Residual test ***************

RESSP11=sqrt(covRES(1,1));          %Calculate theoretical upper error bound on residual for omega1hat
RESSP11P=-RESSP11;                  %Calculate theoretical lower error bound on residual for omega1hat
RESSP22=sqrt(covRES(2,2));          %Calculate theoretical upper error bound on residual for omega2hat
RESSP22P=-RESSP22;                  %Calculate theoretical lower error bound on residual for omega2hat
RESSP33=sqrt(covRES(3,3));          %Calculate theoretical upper error bound on residual for omega3hat
RESSP33P=-RESSP33;                  %Calculate theoretical lower error bound on residual for omega3hat
RESSP44=sqrt(covRES(4,4));          %Calculate theoretical upper error bound on residual for q1hat
RESSP44P=-RESSP44;                  %Calculate theoretical lower error bound on residual for q1hat
RESSP55=sqrt(covRES(5,5));          %Calculate theoretical upper error bound on residual for q2hat
RESSP55P=-RESSP55;                  %Calculate theoretical lower error bound on residual for q2hat
RESSP66=sqrt(covRES(6,6));          %Calculate theoretical upper error bound on residual for q3hat
RESSP66P=-RESSP66;                  %Calculate theoretical lower error bound on residual for q3hat
RESSP77=sqrt(covRES(7,7));          %Calculate theoretical upper error bound on residual for q4hat
RESSP77P=-RESSP77;                  %Calculate theoretical lower error bound on residual for q4hat

ArrayRES1(count+1)=residual(1);
ArrayRES2(count+1)=residual(2);
ArrayRES3(count+1)=residual(3);
ArrayRES4(count+1)=residual(4);
ArrayRES5(count+1)=residual(5);
ArrayRES6(count+1)=residual(6);
ArrayRES7(count+1)=residual(7);

ArrayRESSP11(count+1)=RESSP11;
ArrayRESSP11P(count+1)=RESSP11P;
ArrayRESSP22(count+1)=RESSP22;
ArrayRESSP22P(count+1)=RESSP22P;
ArrayRESSP33(count+1)=RESSP33;
ArrayRESSP33P(count+1)=RESSP33P;
ArrayRESSP44(count+1)=RESSP44;
```

```
  ArrayRESSP44P(count+1)=RESSP44P;
  ArrayRESSP55(count+1)=RESSP55;
  ArrayRESSP55P(count+1)=RESSP55P;
  ArrayRESSP66(count+1)=RESSP66;
  ArrayRESSP66P(count+1)=RESSP66P;
  ArrayRESSP77(count+1)=RESSP77;
  ArrayRESSP77P(count+1)=RESSP77P;

  count=count+1;                                          %increment counter

end

%****** calculate mean square errors for "instantaneous" and for UKF ******
%****** mean square error excludes first 5 samples to reduce effects of transients *****

s2=length(omega1meas);
s1=5;

omega1meas_mse=mean_square_error(omega1meas(s1:s2),omega1(s1:s2));
omega2meas_mse=mean_square_error(omega2meas(s1:s2),omega2(s1:s2));
omega3meas_mse=mean_square_error(omega3meas(s1:s2),omega3(s1:s2));
q1_q_min_mse=mean_square_error(q_min(1,(s1:s2)),q1(s1:s2));
q2_q_min_mse=mean_square_error(q_min(2,(s1:s2)),q2(s1:s2));
q3_q_min_mse=mean_square_error(q_min(3,(s1:s2)),q3(s1:s2));
q4_q_min_mse=mean_square_error(q_min(4,(s1:s2)),q4(s1:s2));

omega1_ukf_mse=mean_square_error(omega1hat(s1:s2),omega1(s1:s2));
omega2_ukf_mse=mean_square_error(omega2hat(s1:s2),omega2(s1:s2));
omega3_ukf_mse=mean_square_error(omega3hat(s1:s2),omega3(s1:s2));
q1_ukf_mse=mean_square_error(q1hat(s1:s2),q1(s1:s2));
q2_ukf_mse=mean_square_error(q2hat(s1:s2),q2(s1:s2));
q3_ukf_mse=mean_square_error(q3hat(s1:s2),q3(s1:s2));
q4_ukf_mse=mean_square_error(q4hat(s1:s2),q4(s1:s2));

%*************** Plotting and output statements *****************

plots=input('Enter "1" to display plots. Enter "0" for no plots. Default is no plots:  ');
if isempty(plots)
  plots=0;
end

if plots==1

%****** plots of error vs. true and theoretical error bounds *******
```

```
plot(ArrayT,(omega1(1:s2)-omega1hat),ArrayT,ArraySP11,'x',ArrayT,ArraySP11P,'x');
grid on;title('error in UKF estimate of omega1');
legend('error in omega1hat','theoretical bound');
xlabel('time (sec)');ylabel('error in omega1hat (rad/s)');
pause;close;clc;

plot(ArrayT,(omega2(1:s2)-omega2hat),ArrayT,ArraySP22,'x',ArrayT,ArraySP22P,'x');
grid on;title('error in UKF estimate of omega2');
legend('error in omega2hat','theoretical bound');
xlabel('time (sec)');ylabel('error in omega2hat (rad/s)');
pause;close;clc;

plot(ArrayT,(omega3(1:s2)-omega3hat),ArrayT,ArraySP33,'x',ArrayT,ArraySP33P,'x');
grid on;title('error in UKF estimate of omega3');
legend('error in omega3hat','theoretical bound');
xlabel('time (sec)');ylabel('error in omega3hat (rad/s)');
pause;close;clc;

plot(ArrayT,(q1(1:s2)-q1hat),ArrayT,ArraySP44,'x',ArrayT,ArraySP44P,'x');
grid on;title('error in UKF estimate of q1');
legend('error in q1hat','theoretical bound');
xlabel('time (sec)');ylabel('error in q1');
pause;close;clc;

plot(ArrayT,(q2(1:s2)-q2hat),ArrayT,ArraySP55,'x',ArrayT,ArraySP55P,'x');
grid on;title('error in UKF estimate of q2');
legend('error in q2hat','theoretical bound');
xlabel('time (sec)');ylabel('error in q2');
pause;close;clc;

plot(ArrayT,(q3(1:s2)-q3hat),ArrayT,ArraySP66,'x',ArrayT,ArraySP66P,'x');
grid on;title('error in UKF estimate of q3');
legend('error in q3hat','theoretical bound');
xlabel('time (sec)');ylabel('error in q3');
pause;close;clc;

plot(ArrayT,(q4(1:s2)-q4hat),ArrayT,ArraySP77,'x',ArrayT,ArraySP77P,'x');
grid on;title('error in UKF estimate of q4');
legend('error in q4hat','theoretical bound');
xlabel('time (sec)');ylabel('error in q4');
pause;close;clc;

%****** plots to show true vs. quaternion derived alone vs. UKF estimates *******
```

```
plot( ArrayT, omega1meas,'gx',ArrayT, omega1hat,'m', ArrayT, omega1(1:s2),'--');grid on;
title('omega1 from sensor, UKF estimate of omega1, "True" omega1' );
xlabel('time (sec)');ylabel('omega1 (rad/s)');
legend('omega1 from rate sensor','omega1 from UKF','true omega1');
pause;clc;close;

plot( ArrayT, omega2meas,'gx',ArrayT, omega2hat,'m', ArrayT, omega2(1:s2),'--');grid on;
title('omega2 from sensor, UKF estimate of omega2, "True" omega2' );
xlabel('time (sec)');ylabel('omega2 (rad/s)');
legend('omega2 from rate sensor','omega2 from UKF','true omega2');
pause;clc;close;

plot( ArrayT, omega3meas,'gx',ArrayT, omega3hat,'m', ArrayT, omega3(1:s2),'--');grid on;
title('omega3 from sensor, UKF estimate of omega3, "True" omega3' );
xlabel('time (sec)');ylabel('omega3 (rad/s)');
legend('omega3 from rate sensor','omega3 from UKF','true omega3');
pause;clc;close;

%****** plots of residual vs. theoretical bounds on residual *******

resplots=input('Enter "1" to display plots from residual test. Enter "0" for no plots. Default is no plots:  ');
if isempty(resplots)
  resplots=0;
end

if resplots==1

  plot(ArrayT,ArrayRES1,'-+',ArrayT,ArrayRESSP11,'x',ArrayT,ArrayRESSP11P,'x');
  grid on;title('residual for omega1');
  legend('residual for omega1','theoretical bound');
  xlabel('time (sec)');ylabel('residual for omega1 (rad/s)');
  pause;close;clc;

  plot(ArrayT,ArrayRES2,'-+',ArrayT,ArrayRESSP22,'x',ArrayT,ArrayRESSP22P,'x');
  grid on;title('residual for omega2');
  legend('residual for omega2','theoretical bound');
  xlabel('time (sec)');ylabel('residual for omega2 (rad/s)');
  pause;close;clc;

  plot(ArrayT,ArrayRES3,'-+',ArrayT,ArrayRESSP33,'x',ArrayT,ArrayRESSP33P,'x');
  grid on;title('residual for omega3');
  legend('residual for omega3','theoretical bound');
  xlabel('time (sec)');ylabel('residual for omega3 (rad/s)');
```

```
    pause;close;clc;

    plot(ArrayT,ArrayRES4,'-+',ArrayT,ArrayRESSP44,'x',ArrayT,ArrayRESSP44P,'x');
    grid on;title('residual for q1');
    legend('residual for q1','theoretical bound');
    xlabel('time (sec)');ylabel('residual for q1');
    pause;close;clc;

    plot(ArrayT,ArrayRES5,'-+',ArrayT,ArrayRESSP55,'x',ArrayT,ArrayRESSP55P,'x');
    grid on;title('residual for q2');
    legend('residual for q2','theoretical bound');
    xlabel('time (sec)');ylabel('residual for q2');
    pause;close;clc;

    plot(ArrayT,ArrayRES6,'-+',ArrayT,ArrayRESSP66,'x',ArrayT,ArrayRESSP66P,'x');
    grid on;title('residual for q3');
    legend('residual for q3','theoretical bound');
    xlabel('time (sec)');ylabel('residual for q3');
    pause;close;clc;

    plot(ArrayT,ArrayRES7,'-+',ArrayT,ArrayRESSP77,'x',ArrayT,ArrayRESSP77P,'x');
    grid on;title('residual for q4');
    legend('residual for q4','theoretical bound');
    xlabel('time (sec)');ylabel('residual for q4 (rad/s)');
    pause;close;clc;

end

end

'Summary of performance of each method...'
' '
'The MSE in omega1 from the rate sensor is (rad/s): ',' ',omega1meas_mse
' '
'The MSE in omega1 for the UKF estimates is (rad/s): ',' ',omega1_ukf_mse
' '
'The MSE in omega2 derived  from the rate sensor is (rad/s): ',' ',omega2meas_mse
' '
'The MSE in omega2 for the UKF estimates is (rad/s): ',' ',omega2_ukf_mse
' '
'Please press "return" to continue...'
pause;clc;close;

'The MSE in omega3 derived  from the rate sensor is  (rad/s): ',' ',omega3meas_mse
```

```
' '
'The MSE in omega3 for the UKF estimates is (rad/s): ',' ',omega3_ukf_mse
' '
'The MSE in q1 derived using Gauss-Newton is: ',' ',q1_q_min_mse
' '
'The MSE in q1 for the UKF estimates is: ',' ',q1_ukf_mse
' '
'Please press "return" to continue...'
pause;clc;close;

'Summary of performance of each method (cont.) ...'
' '
'The MSE in q2 derived using Gauss-Newton is: ',' ',q2_q_min_mse
' '
'The MSE in q2 for the UKF estimates is: ',' ',q2_ukf_mse
' '
'press "return" to continue...'
pause;clc;close;

'Summary of performance of each method (cont.) ...'
' '
'The MSE in q3 derived using Gauss-Newton is: ',' ',q3_q_min_mse
' '
'The MSE in q3 for the UKF estimates is: ',' ',q3_ukf_mse
' '
'The MSE in q4 derived using Gauss-Newton is: ',' ',q4_q_min_mse
' '
'The MSE in q4 for the UKF estimates is: ',' ',q4_ukf_mse
' '
'press "return" to continue...'
pause;clc;close;

'Summary of performance of each method (cont.) ...'

%****** rotate each vector using actual and UKF attitude quaternion ******
%****** check mean-square-error between vectors ******

%'****** Evaluate Mean-Square-Error of rotated vectors using Gauss-Newton derived quaternion ******'
%'Press "return" to rotate the reference vector at each time step to the'
%'representation in the rotated frame using both the actual rotation quaternion'
%'and the quaternion derived using the UKF...once rotation is complete,'
%'calculate the mean-square-error between the results.'
%pause;close;clc;
```

```
qukf=[q1hat;q2hat;q3hat;q4hat];                              %matrix representation of ukf quaternions
%for j=1:length(q1hat)
%    qukf(1:4,j)=qukf(1:4,j)/norm(qukf(1:4,j));              %normalize to insure unit quaternion
%end

ukf_mse_error=[ ];
for j=1:count;
   ukfvec(1:6,j)=qframerot6(qukf(1:4,j),vref(1:6,j));        %rotate ref vector using min error quaternion
   Q_vec(1:6,j)=qframerot6(Qvec(1:4,j),vref(1:6,j));         %rotate ref vector using true quaternion
   ukf_mse_error(j)=mean_square_error(ukfvec(1:6,j),Q_vec(1:6,j)); %calculate MSE between rotated vectors
end

%***** determine angle between rotated vectors *****
for k=1:count
   oukfsunangle(k)=acos(dot(ukfvec(1:3,k),Q_vec(1:3,k))/(norm(ukfvec(1:3,k))*norm(Q_vec(1:3,k))));
   oukfmagangle(k)=acos(dot(ukfvec(4:6,k),Q_vec(4:6,k))/(norm(ukfvec(4:6,k))*norm(Q_vec(4:6,k))));
   oGNsunangle(k)=acos(dot(q_minvec(1:3,k),Q_vec(1:3,k))/(norm(q_minvec(1:3,k))*norm(Q_vec(1:3,k))));
   oGNmagangle(k)=acos(dot(q_minvec(4:6,k),Q_vec(4:6,k))/(norm(q_minvec(4:6,k))*norm(Q_vec(4:6,k))));
end

oukfsunangle=oukfsunangle*180/pi;
oukfmagangle=oukfmagangle*180/pi;
oGNsunangle=oGNsunangle*180/pi;
oGNmagangle=oGNmagangle*180/pi;

%****** Plot vectors rotated using derived and actual quaternions *****
plot3(Q_vec(1,1:count),Q_vec(2,1:count),Q_vec(3,1:count),'b');grid on;hold on;
plot3(ukfvec(1,1:count),ukfvec(2,1:count),ukfvec(3,1:count),'m');
plot3(Q_vec(4,1:count),Q_vec(5,1:count),Q_vec(6,1:count),'r');grid on;hold on;
plot3(ukfvec(4,1:count),ukfvec(5,1:count),ukfvec(6,1:count),'g');
xlabel('X component');ylabel('Y component');zlabel('Z component');
legend('sun vector, actual q','sun vector, ukf q','mag field vector, actual q', 'mag field vector, ukf q');
title('3-D plot of reference vectors rotated using actual quaternions and ukf derived quaternions');
pause;close;clc;

%****** Plot mean-square-error between reference vector rotated using actual quaternion ******
%****** and reference vector rotated using derived quaternion.                          ******

plot(T(1:count),ukf_mse_error(1:count)),grid on;
title('MSE between reference vectors rotated using ukf derived and actual quaternions');
xlabel('time (sec)');ylabel('MSE');legend('MSE')

'press "return" to continue...'
pause;close;clc;
```

```
'********************* Overall Performance Indicator ************************'
'****** Excludes first five measurements to reduce effect of transients ******'
' '
'The average of the omega mean-square-errors for the differencing method is: '
GNomegaperf=(omega1meas_mse+omega2meas_mse+omega3meas_mse)/3
' '
'The average of the omega mean-square-errors for the ukf results is: '
ukfomegaperf=(omega1_ukf_mse+omega2_ukf_mse+omega3_ukf_mse)/3
' '
'The average of the quaternion component mean-square-errors for the Gauss-Newton method is: '
GNqperf=(q1_q_min_mse+q2_q_min_mse+q3_q_min_mse+q4_q_min_mse)/4
' '
'The average of the quaternion component mean-square-errors for the ukf results is: '
ukfqperf=(q1_ukf_mse+q2_ukf_mse+q3_ukf_mse+q4_ukf_mse)/4
' '
'press "return" to continue...'
pause;clc;close;

'Summary of performance of each method (cont.) ...'
' '
'The average MSE for the ref vectors rotated using GN quaternions: '
mean_GN_mse_error=mean(GN_mse_error)
' '
'The average MSE for the ref vectors rotated using ukf quaternions: '
mean_ukf_mse_error=mean(ukf_mse_error)
' '
'press "return" to continue...'
pause;clc;close;

'Summary of performance of each method (cont.) ...'
' '
'overall GN performance: sqrt[(mean omega MSE)^2 + (mean rotated vector MSE)^2)]'
overall_GN_perf=sqrt(GNomegaperf^2+mean_GN_mse_error^2)
' '
'overall ukf performance: sqrt[(mean omega MSE)^2 + (mean rotated vector MSE)^2)]'
overall_ukf_perf=sqrt(ukfomegaperf^2+mean_ukf_mse_error^2)
' '
%'*************************************************************************************'
%'********************************* ukf summary **************************************'
%'*************************************************************************************'
'omega:       ',ukfomegaperf
'quaternion:  ',mean_ukf_mse_error
'overall:     ',overall_ukf_perf
```

```
%'mean sun angle error:   ',mean(oukfsunangle)
%'std sun angle error:   ',std(oukfsunangle)
%'max sun angle error:   ',max(oukfsunangle)
%'mean mag angle error:   ',mean(oukfmagangle)
%'std mag angle error:   ',std(oukfmagangle)
%'max mag angle error:   ',max(oukfmagangle)
' '
'press "return" to continue...'
pause;close;clc;
'The next option allows the calculation of "error angles" between the body frame rotated using'
'the true attitude quaternion and using the filter estimated quaternion.  This option also allows'
'3D depiction of the body frame, in inertial cooridnates, over time.  It can take a while to generate.'
' '
angles=input('Enter "1" to calculate attitude error angles. Enter "0" for no angles. Default is no angles:  ');
if isempty(angles)
  angles=0;
end

if angles==1

   %Calculate angles between body frame rotated using estimated quaternion and
   %rotated using "true" quaternion

   [ovec1,ovec2,ovec3,ovec1hat,ovec2hat,ovec3hat,ang1,ang2,ang3]=orientation3(Qvec, qukf);

   plot(ArrayT,ang1);grid on;title('Error Angle For Axis 1');xlabel('time (sec)');ylabel('Angle 1 (deg)');
   pause;close;clc;
   plot(ArrayT(2:length(ang2)),ang2(2:length(ang2)));grid on;title('Error Angle For Axis 2');xlabel('time (sec)');ylabel('Angle 2
(deg)');
   pause;close;clc;
   plot(ArrayT,ang3);grid on;title('Error Angle For Axis 3');xlabel('time (sec)');ylabel('Angle 3 (deg)');
   pause;close;clc;

end
```

A.17 Measadjust.m

This script takes attitude_sim.m generated data and allows a user to specify a different measurement interval. It initializes the appropriate arrays and variables, performs the Gauss-Newton error minimization step, and allows the m-files with the filtering algorithms to be run directly, on the reduced set of measurements.

```
%Measadjust - this script decreases number of measurements available for
%filtering.  Prompts user for the measurement interval desired, and then
%extracts these measurements from the original set generated by
%attitude_sim.m.  Reinitializes the appropriate variables and arrays to
%allow successful processing of the reduced measurement set.  Performs
%Gauss-Newton error minimization step.  User must then run norates_ekf.m,
%norates_ukf.m, rates_ekf.m or rates_ukf.m independently to process reduced
%measurement set.
%
%Written by Mark Charlton.  Last updated 31 October 2003.

clc

TSold=TS;
countold=count;

TS=input('Please enter desired interval for measurements in seconds:  ')

if isempty (TS)
   TS=TSold;
end

countstep=ceil(TS/TSold);

count=ceil(count/countstep);

sunmeas=sunmeas(1:3,1:countstep:countold);
magmeas=magmeas(1:3,1:countstep:countold);
SUNVEC=SUNVEC(1:3,1:countstep:countold);
MAGVEC=MAGVEC(1:3,1:countstep:countold);
omega1meas=omega1meas(1:countstep:countold);
omega2meas=omega2meas(1:countstep:countold);
omega3meas=omega3meas(1:countstep:countold);
omega1=omega1(1:countstep:countold);
omega2=omega2(1:countstep:countold);
omega3=omega3(1:countstep:countold);
T=T(1:countstep:countold);
omega1_q_min=[ ];
omega2_q_min=[ ];
omega3_q_min=[ ];
Qvec=Qvec(1:4,1:countstep:countold);
Q_refvec=[ ];
q_minvec=[ ];
```

```
q1=q1(1:countstep:countold);
q2=q2(1:countstep:countold);
q3=q3(1:countstep:countold);
q4=q4(1:countstep:countold);

'****** Gauss-Newton error minimization ******'
'Press "return" to calculate the attitude quaternion from the noisy'
'attitude measurements using a Gauss-Newton error minimization...'
pause;close;clc;

vref=[ ];
vrot=[ ];
q_min=[ ];
q_min_dot=[ ];
qmin1init=0;                                    %initial estimate of attitude quaternion
qmin2init=0;
qmin3init=0;
qmin4init=1;
A=[ ];
check=[ ];
err=[ ];

for i=1:count;
  vref(1:6,i)=[SUNVEC(1:3,i);MAGVEC(1:3,i)];    %form reference vector from known inertial vectors

  if i<2
    q_init=[qmin1init qmin2init qmin3init qmin4init]';
  else
    q_init=q_min(1:4,i-1);
  end

  if norm(sunmeas(1:3,i))<.01             %default values in case of loss of data (no sun vector measurement)
    if i==1
      q_min(1:4,i)=[qmin1init qmin2init qmin3init qmin4init]';
    else
      q_min(1:4,i)=qukf(1:4,i-1);
    end

    A(1:4,1:6,i)=A(1:4,1:6,i-1);
    check(i)=1;                                 %check = 1 indicates missing measurement
    err(i)=1;                                   %err = 1 indicates missing measurement;

    omega1meas(i)=omega1hat(i-1);
    omega2meas(i)=omega2hat(i-1);
    omega3meas(i)=omega3hat(i-1);
  elseif norm(magmeas(1:3,i))<.01                 %default values in case of loss of data (no mag field measurement)
          if i==1
            q_min(1:4,i)=[qmin1init qmin2init qmin3init qmin4init]';
          else
            q_min(1:4,i)=qukf(1:4,i-1);
          end
          A(1:4,1:6,i)=A(1:4,1:6,i-1);
          check(i)=1;                                       %check = 1 indicates missing measurement
          err(i)=1;                                         %err = 1 indicates missing measurement;

          omega1meas(i)=omega1hat(i-1);
          omega2meas(i)=omega2hat(i-1);
          omega3meas(i)=omega3hat(i-1);

        else
          %if measurements exist...proceed with minimization...

          unitsunmeas(1:3,i)=sunmeas(1:3,i)/norm(sunmeas(1:3,i));          %insure noisy sun measurement is unit vector
          unitmagmeas(1:3,i)=magmeas(1:3,i)/norm(magmeas(1:3,i));          %insure noisy mag measurement is unit vector

          vrot(1:6,i)=[unitsunmeas(1:3,i);unitmagmeas(1:3,i)];             %form vector to be rotated by "best" quaternion

          steps=10;                               %defines max allowable iterations for convergence within Gauss_newton
          tol=.01;                                                   %defines convergence tolerance for Gauss_Newton

          [q_min(1:4,i), A(1:4,1:6,i), check(i), err(i)]=Gauss_newton(vref(1:6,i), vrot(1:6,i), q_init, steps, tol);

        end
      end

      %****** Calculate standard deviation of error in measured omegas to use in filter R matrices -- not available in real world
*******

      stdomega1meas=std(omega1measerror);
      stdomega2meas=std(omega2measerror);                            %use these when rate measurements are available
      stdomega3meas=std(omega3measerror);

      %****** rotate each vector using actual and derived attitude quaternion ******

      %****** check mean-square-error between vectors ******

      '****** Evaluate Mean-Square-Error of rotated vectors using Gauss-Newton derived quaternion ******'
      'Press "return" to rotate the reference vector at each time step to the'
    'representation in the rotated frame using both the actual rotation quaternion'
    'and the quaternion derived from the noisy measurements...once rotation is complete,'
    'calculate the mean-square-error between the results.'
    pause;close;clc;

    GN_mse_error=[ ];
    q_minvec=[ ];
    Q_refvec=[ ];

    for j=1:count;
      Q_refvec(1:6,j)=qframerot6(Qvec(1:4,j),vref(1:6,j));              %rotate ref vector using true quaternion
      q_minvec(1:6,j)=qframerot6(q_min(1:4,j),vref(1:6,j));             %rotate ref vector using min error quaternion
      GN_mse_error(j)=mean_square_error(q_minvec(1:6,j),Q_refvec(1:6,j)); %calculate mse between rotated vectors(exlude
first 5)
    end

    'The average mse error for the reference vectors rotated using the GN-derived quaternion is: '
    mean_GN_mse_error=mean(GN_mse_error)

    %****** Plot mean-square-error between reference vector rotated using actual quaternion ******
    %****** and reference vector rotated using derived quaternion.                        ******

    plot(T(1:count),GN_mse_error(1:count)),grid on;
    title('MSE between reference vectors rotated using derived and actual quaternions');
    xlabel('time (sec)');ylabel('MSE');legend('MSE')

    'press "enter" to continue...'
    pause;close;clc;

%end of program
```

Appendix B
Representative Plots Available From Data Simulation

This appendix contains plots representative of those generated by attitude_sim.m. Many additional numerical outputs are displayed on-screen, and the display of plots is user-selectable. The plots in this appendix were generated for a case similar to the baseline simulation. The only difference is that a duration of 10 seconds is used instead of 40 seconds in order to make the plots more clear for insertion in this text. This simulation uses the baseline sensor measurement standard deviations of 1.333 deg in each axis for the Sun sensor, 3.333 deg in each axis for the magnetometer, and 0.333 rev/min for the rate sensor. The rate profile in this case asymptotically approaches the final values of 0.5 rev/min about the body x-axis, 0.5 rev/min about the body y-axis, and 225 rev/min about the body z-axis and the final rates are achieved at 5 seconds.

Figures B.1 through B.3 illustrate the generation of simulated Sun vector components and the noisy measurements of those components. This representation can be though of as how the vector "looks" from the rocket-mounted body frame as opposed to the vector in inertial coordinates.

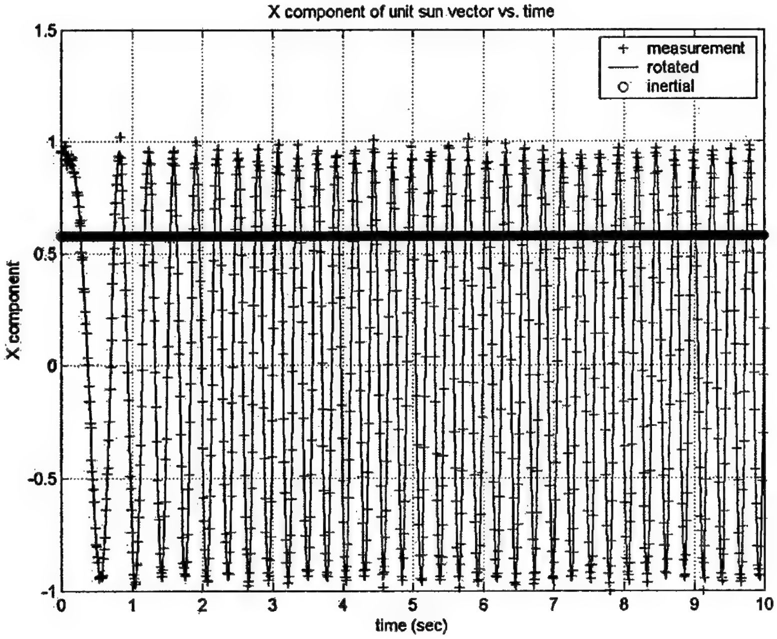


Figure B.1: Simulated "x" Component of Sun Vector – Inertial, Rotated, Measured

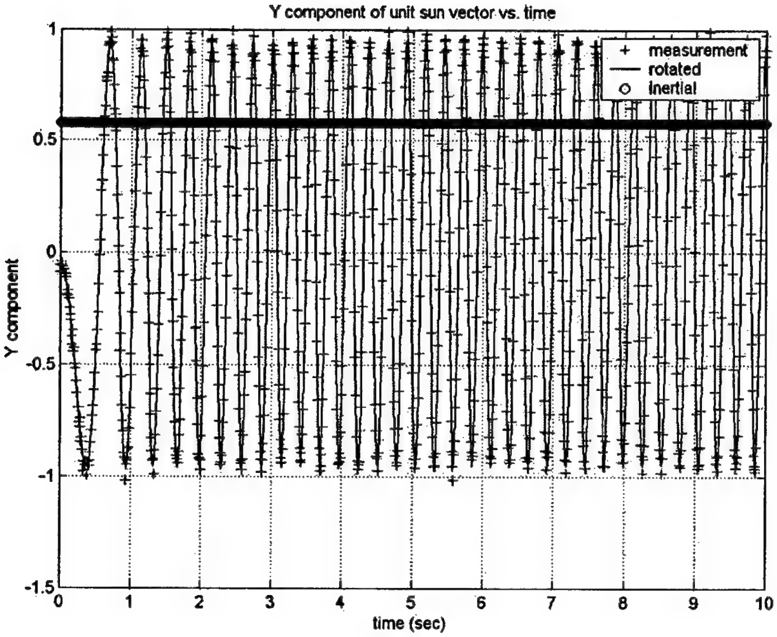


Figure B.2: Simulated "y" Component of Sun Vector – Inertial, Rotated, Measured

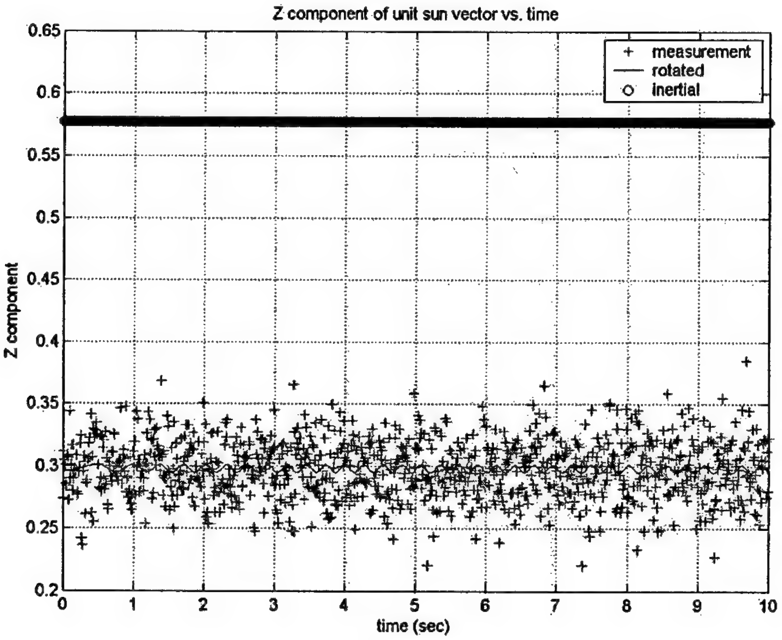


Figure B.3: Simulated "z" Component of Sun Vector – Inertial, Rotated, Measured

The next figure is a three dimensional depiction of these three vector components of the rotated vector over time and the inertial vector.

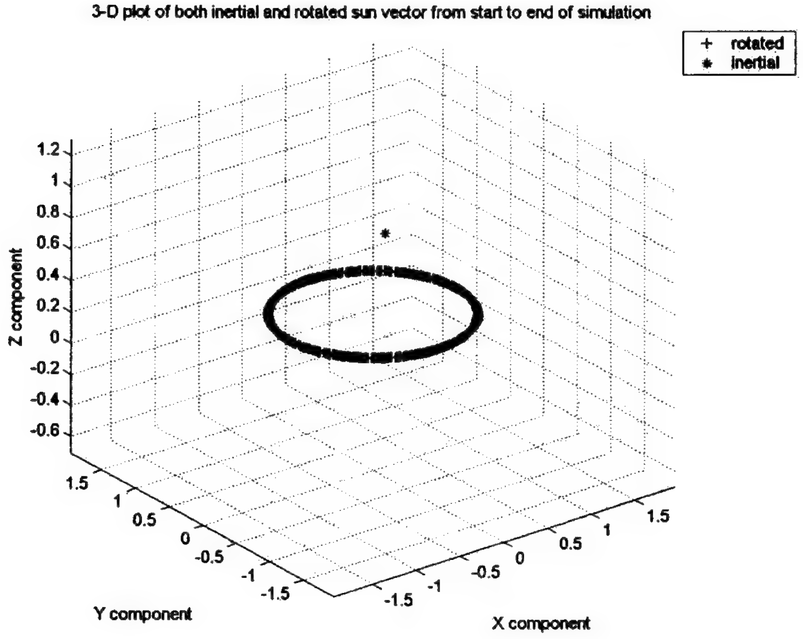


Figure B.4: Three Dimensional Depiction of Inertial and Rotated Sun Vector

The four figures that follow are a representation of the same information for the magnetic field vector. For both the Sun and magnetic field vectors, not only is the "true" rotated vector generated, but also the simulated noisy measurement upon which the filter algorithms operate. The noise statistics are defined by the user when running attitude_sim.m. Figure B.9 is a three dimensional view of both the Sun and magnetic field vectors, both rotated and inertial. It also displays the "vector" part of the rotation quaternion at each time step. Figures B.10 through B.12 illustrate the simulated body rotational rate profiles as well as the simulated noisy measurement of the rate about each body axis at each time step.

In addition to simulating the sensed vectors, attitude_sim.m propagates the Euler angles forward in time and then converts these Euler angles to an attitude quaternion at each time step. This provides the means to generate a "truth" with which to compare the filter estimates to assess performance. Figure B.13 depicts the Euler angle rates, derived from the body rate profiles. This is followed by Figure B.14, the time history of the Euler angles, and finally by the time history of the attitude quaternion derived from the Euler angles in Figure B.15.

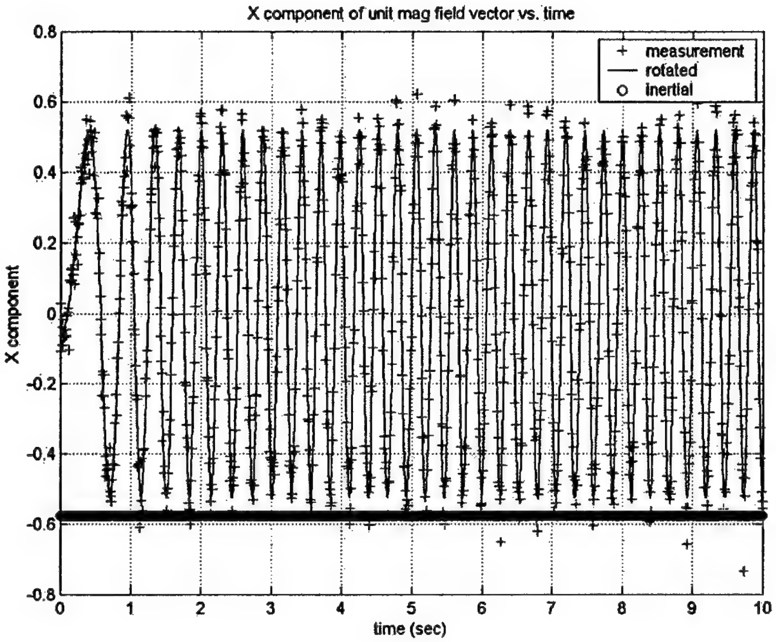


Figure B.5: Simulated "x" Component of Magnetic Field Vector – Inertial, Rotated, Measured

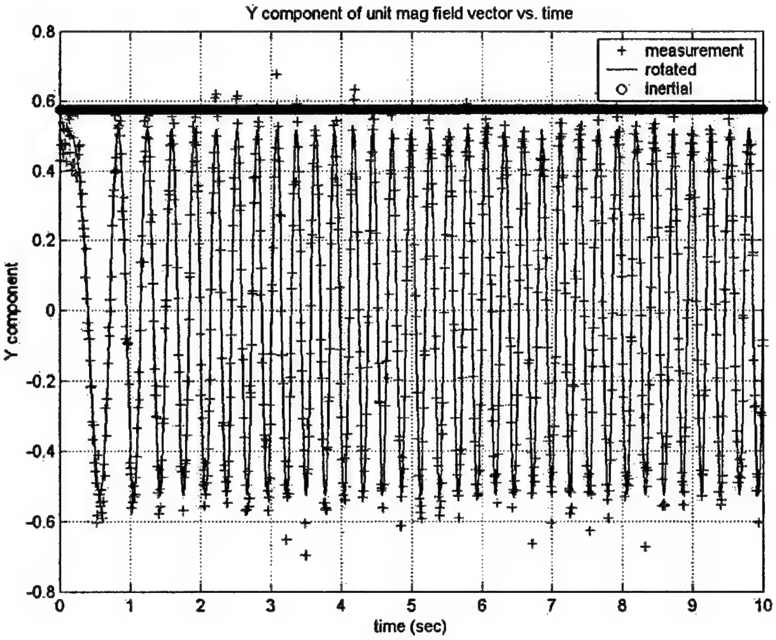


Figure B.6: Simulated "y" Component of Magnetic Field Vector – Inertial, Rotated, Measured

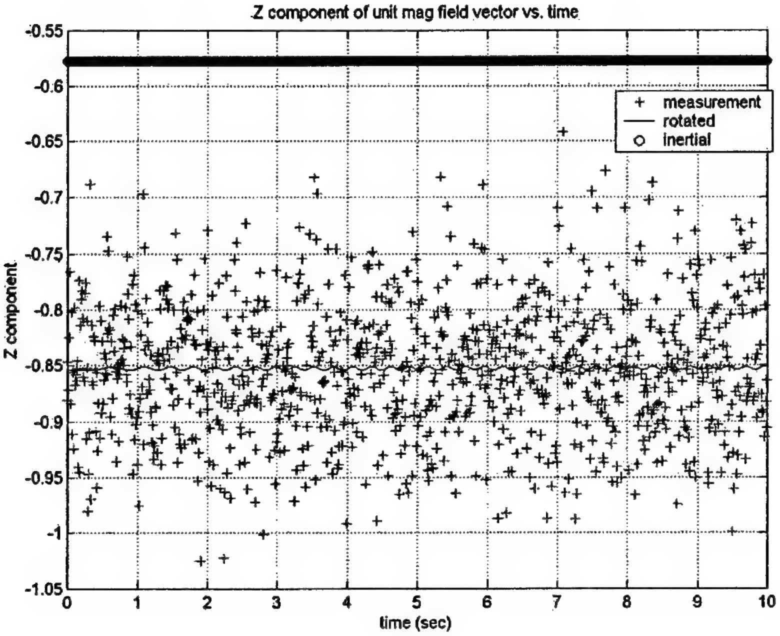


Figure B.7: Simulated "z" Component of Magnetic Field Vector – Inertial, Rotated, Measured

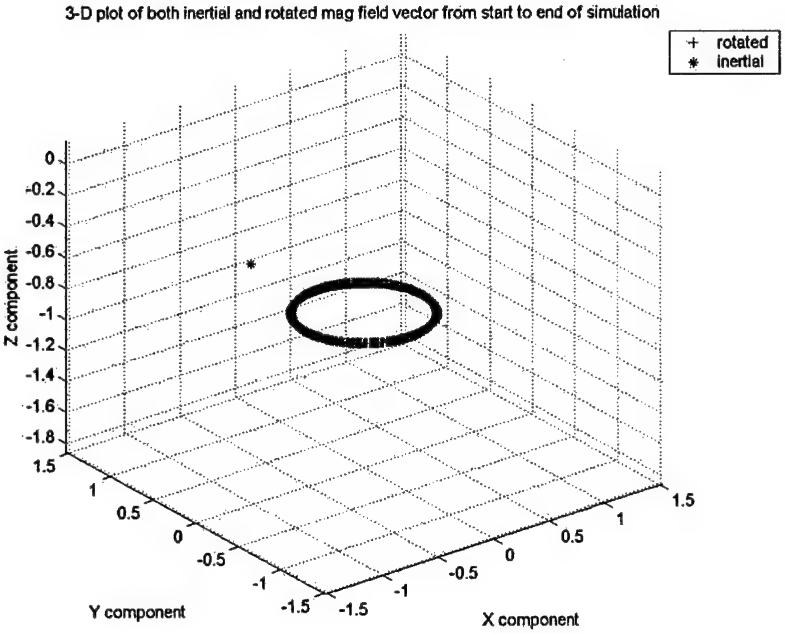


Figure B.8: Three Dimensional Depiction of Inertial and Rotated Magnetic Field Vector

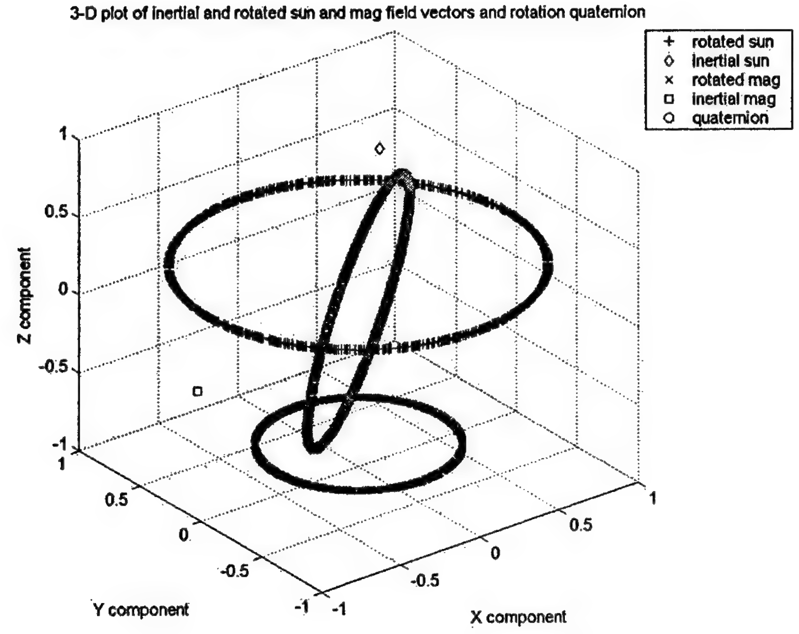


Figure B.9: Three Dimensional View of Inertial and Rotated Reference Vectors and Quaternion

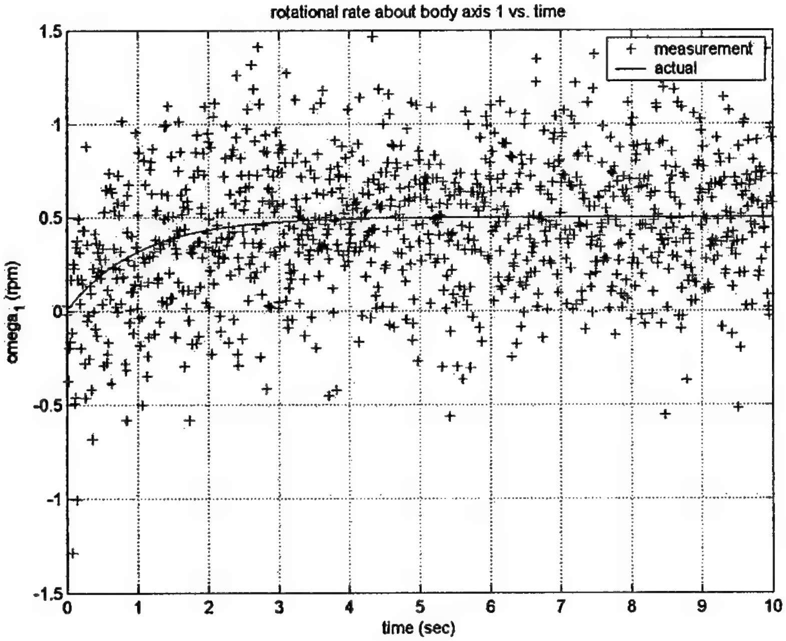


Figure B.10: Simulated ω_1 Profile and Noisy Measurements

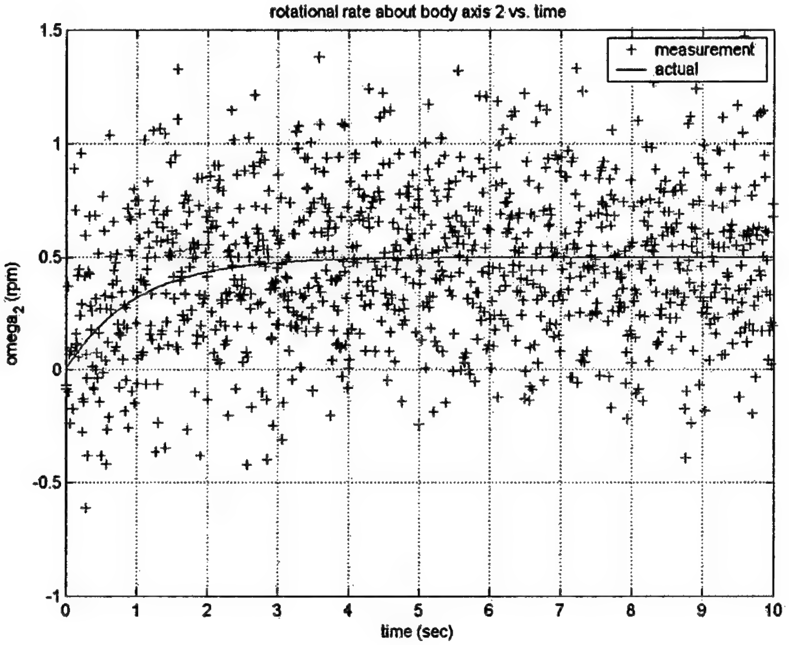


Figure B.11: Simulated ω_2 Profile and Noisy Measurements

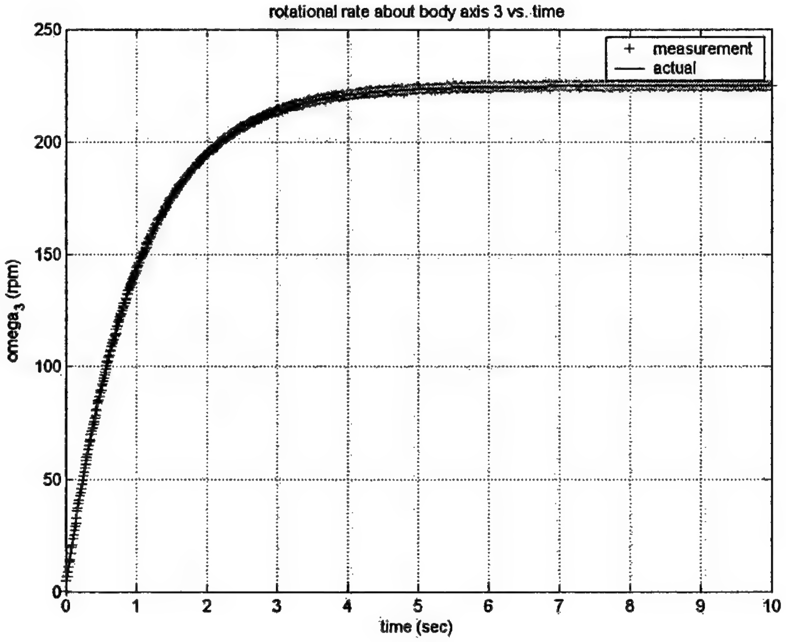


Figure B.12: Simulated ω_3 Profile and Noisy Measurements

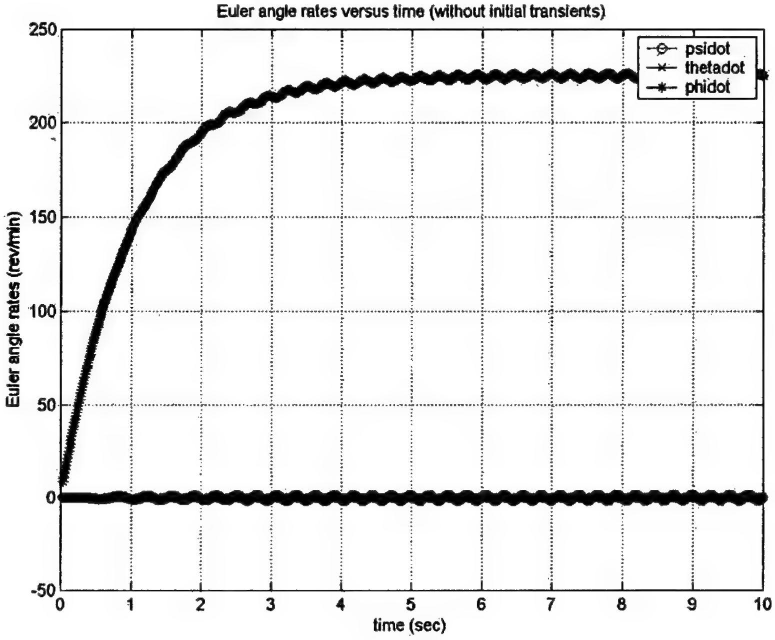


Figure B.13: Simulated Euler Angular Rate Profile

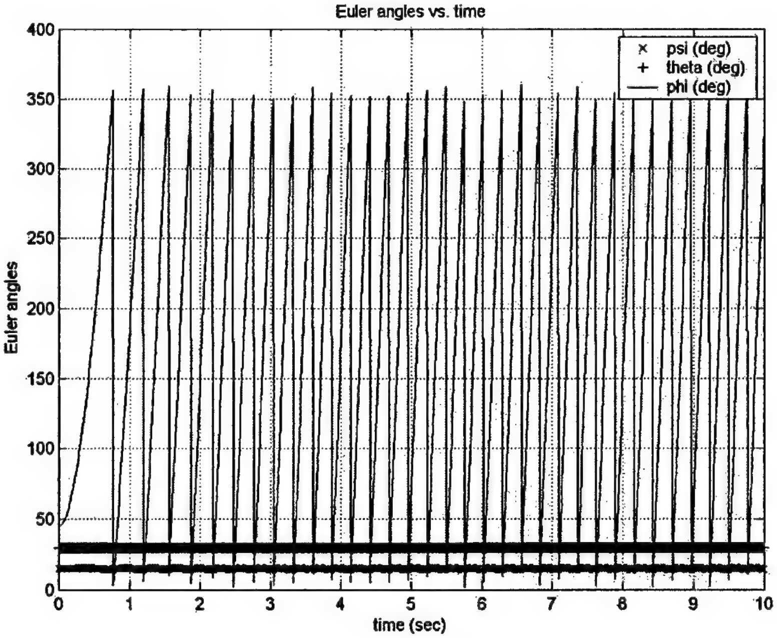


Figure B.14: Simulated Euler Angle History

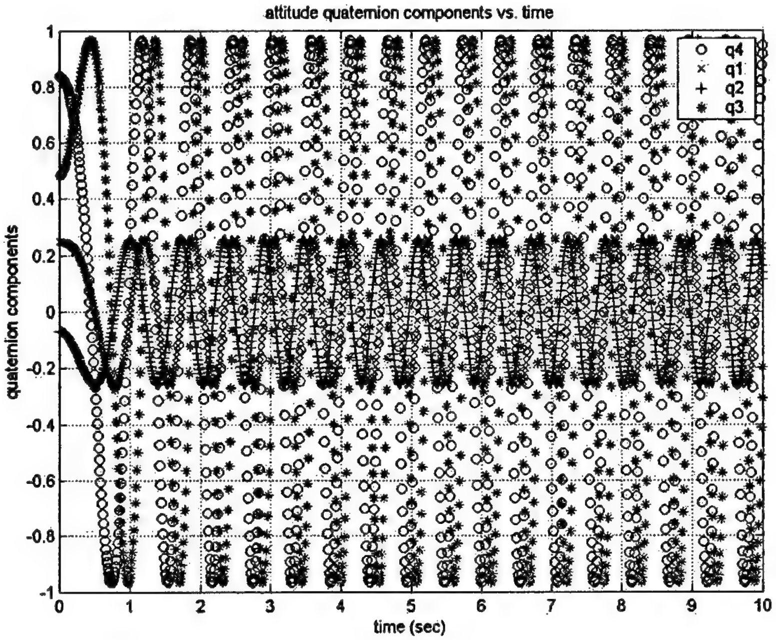


Figure B.15: Simulated Attitude Quaternion History

Appendix C

Representative Plots Available from Filter Algorithms

This appendix contains plots representative of those generated by attitude_filter.m, norates_ekf.m, rates_ekf.m, norates_ukf.m and rates_ukf.m. Many additional numerical outputs are displayed on-screen, and the display of plots is user-selectable. The plots in this appendix were generated for a case similar to the baseline simulation. The only difference is that a duration of 10 seconds is used instead of 40 seconds in order to make the plots more clear for insertion in this text. This simulation uses the baseline sensor measurement standard deviations of 1.333 deg in each axis for the Sun sensor, 3.333 deg in each axis for the magnetometer, and 0.333 rev/min for the rate sensor. The rate profile used asymptotically approaches the final values of 0.5 rev/min about the body x-axis, 0.5 rev/min about the body y-axis, and 225 rev/min about the body z-axis and the final rates are achieved at 5 seconds.

The first four figures depict the Gauss-Newton derived quaternion components versus the "true" quaternions generated from the attitude_sim.m propagated Euler angles. The solid lines are the true values and the "+" symbols are the components derived from the error-minimization step.

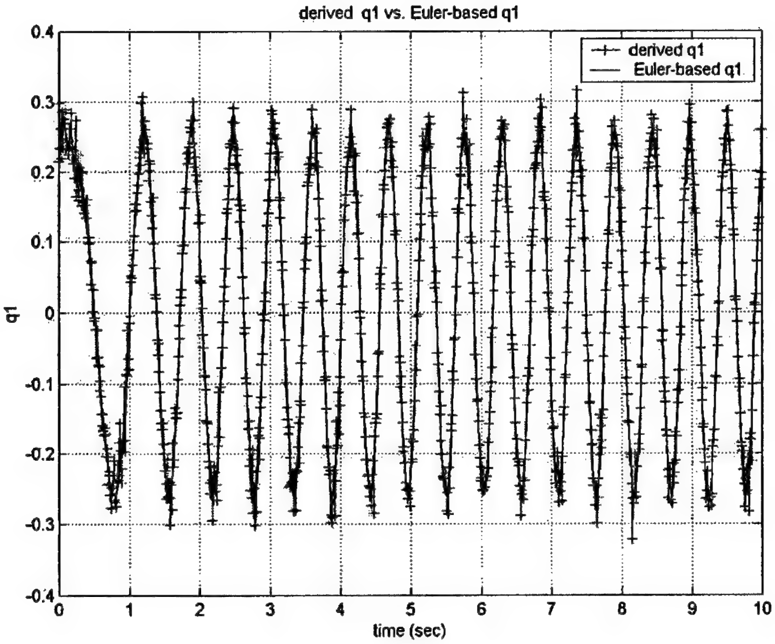


Figure C.1: Gauss-Newton Derived q_1 Versus True q_1

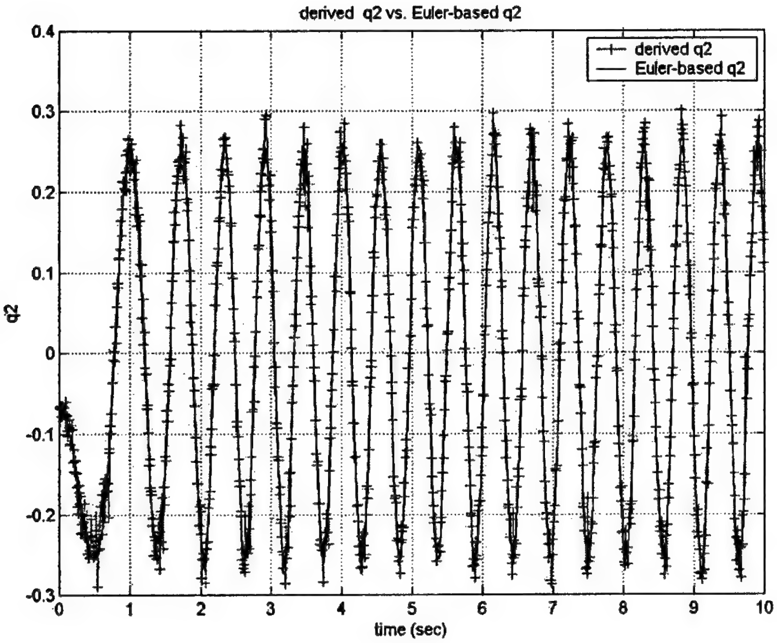


Figure C.2: Gauss-Newton Derived q_2 Versus True q_2

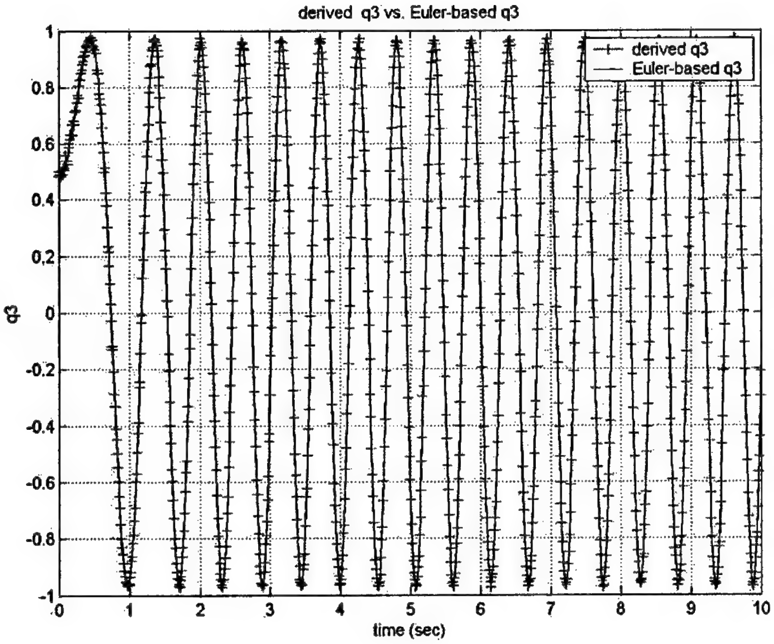


Figure C.3: Gauss-Newton Derived q_3 Versus True q_3

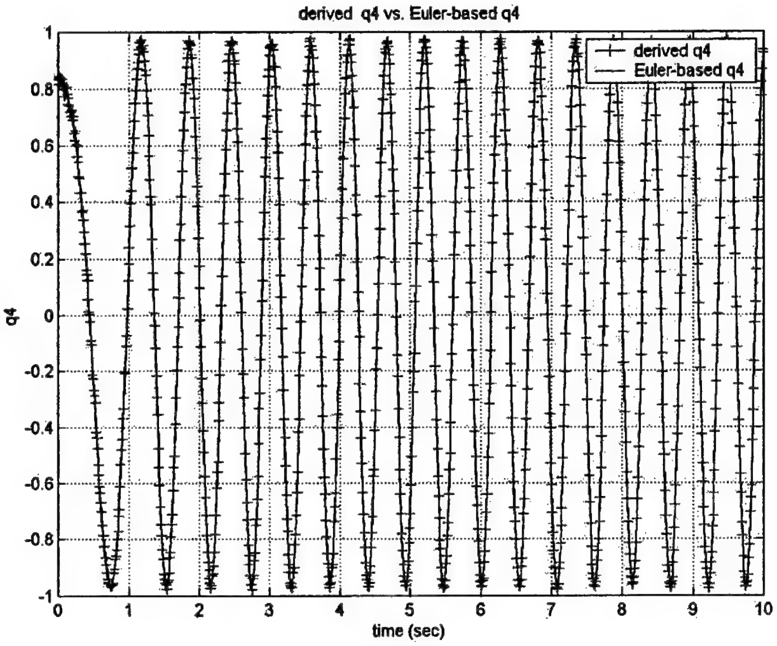


Figure C.4: Gauss-Newton Derived q_4 Versus True q_4

Figure C.5 is a three-dimensional depiction of the reference vector rotated using the actual quaternion from attitude_sim.m and using the quaternion from the Gauss-Newton error minimization. The next figure gives an illustration of the mean square error at each time step between these two rotated vectors. This is a visual cue to how well the error minimization is performing. These two figures are then followed by plots of the error in the UKF estimates of the rotational rates. The theoretical bounds on the error come from the diagonal element of the covariance matrix that corresponds to that state. One good indication of proper filter performance for a Kalman style filter is that at least 68% of the error estimates fall within the theoretical bounds [17]. Following are figures showing the error in the UKF estimate of each quaternion component, again with the corresponding theoretical bounds. Next, the filter estimate of each rate, its "measurements" and the true rotational rate are plotted for each axis in Figures C.14 through C.16. These give a visual indication of the quality of the UKF rate estimates. For a similar visual assessment of the quality of the quaternion estimate, the next figure shows the reference vectors rotated using the UKF derived quaternion versus that rotated using the true quaternion and Figure C.18 displays the mean square error between the vectors rotated using these two quaternions. The final figure depicts the body frame unit vectors rotated in the inertial frame using both the true and the filter generated quaternions. This gives a visual interpretation of the "pointing accuracy" error of the solution. These plots were generated using the

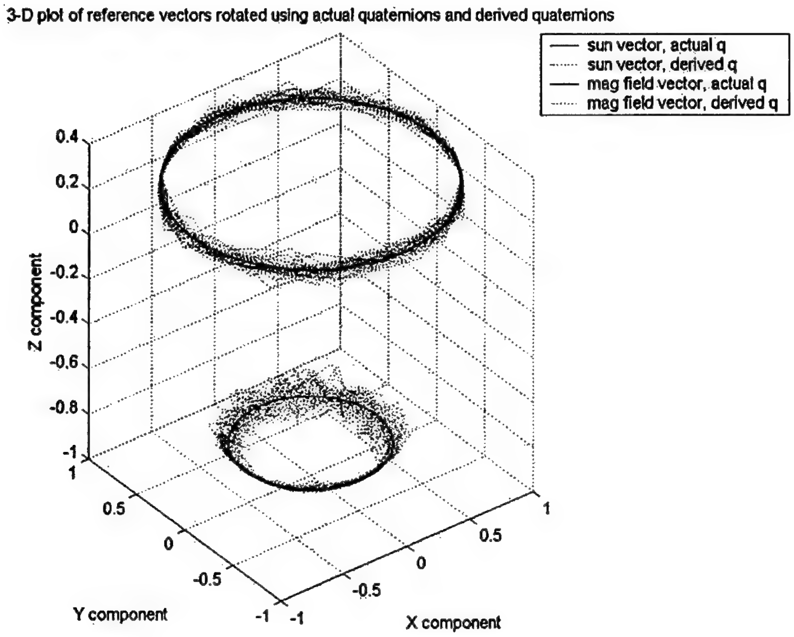


Figure C.5: Reference Vectors Rotated Using Gauss-Newton Derived and True Quaternions

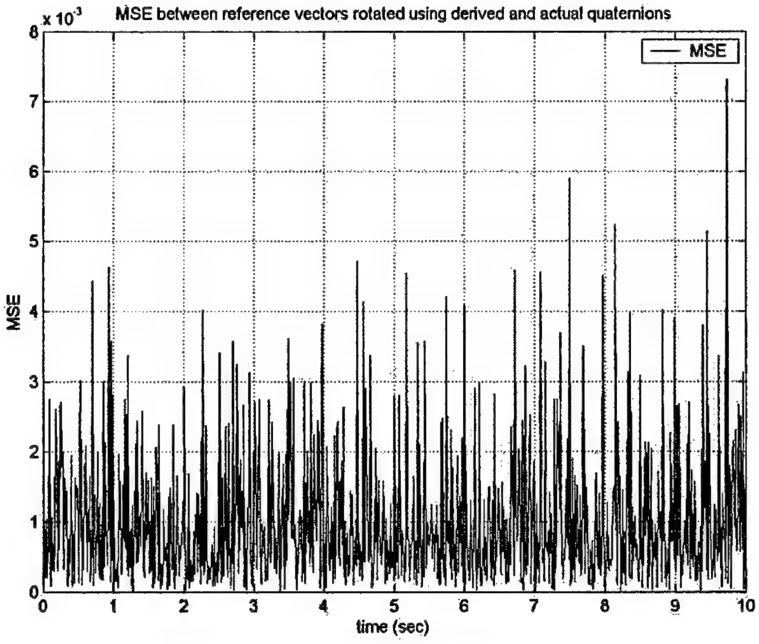


Figure C.6: MSE Between Vectors Rotated Using Gauss-Newton and True Quaternions

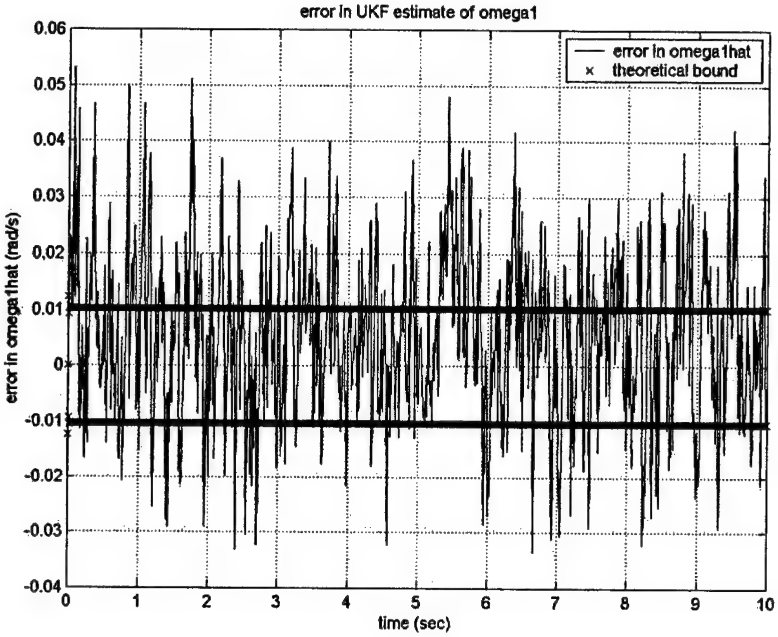


Figure C.7: Error in the UKF Estimate of the Rotational Rate About the “x” Body Axis

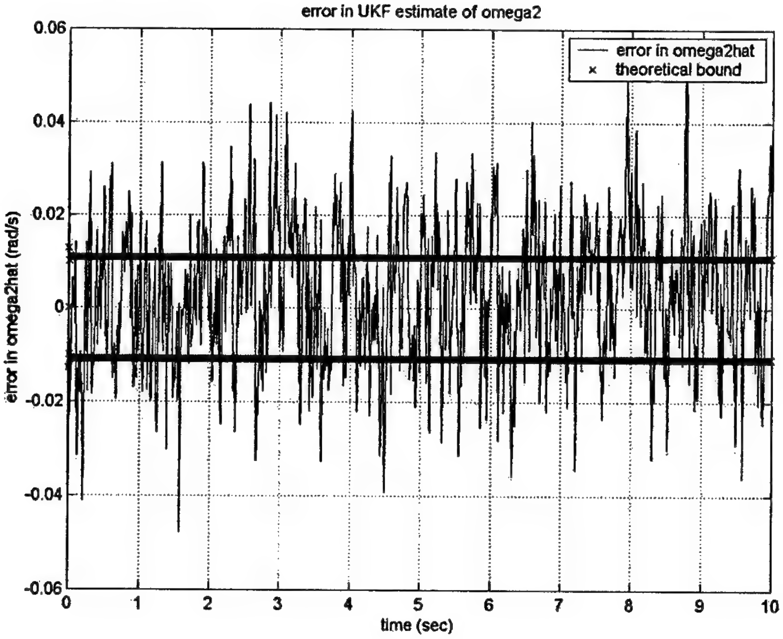


Figure C.8: Error in the UKF Estimate of the Rotational Rate About the “y” Body Axis

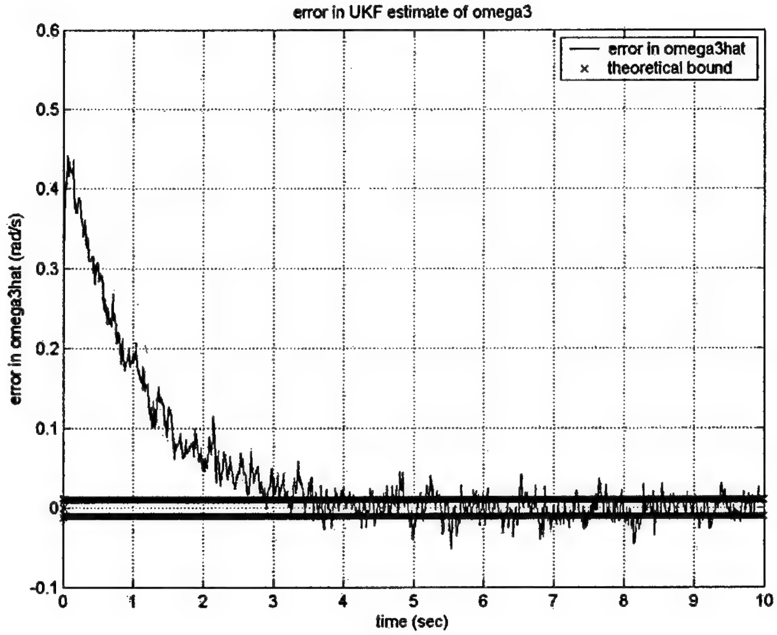


Figure C.9: Error in the UKF Estimate of the Rotational Rate About the "z" Body Axis

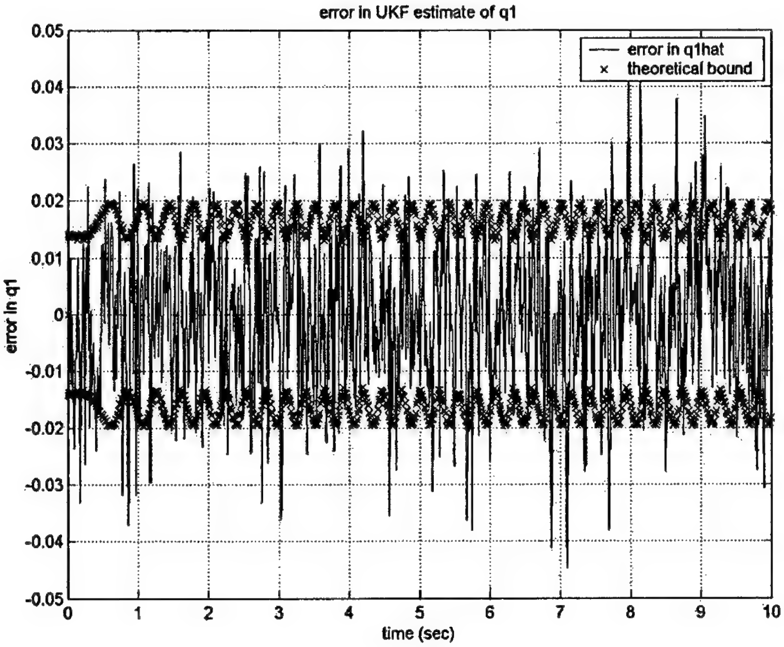


Figure C.10: Error in the UKF Estimate of the q_1 Component of the Quaternion

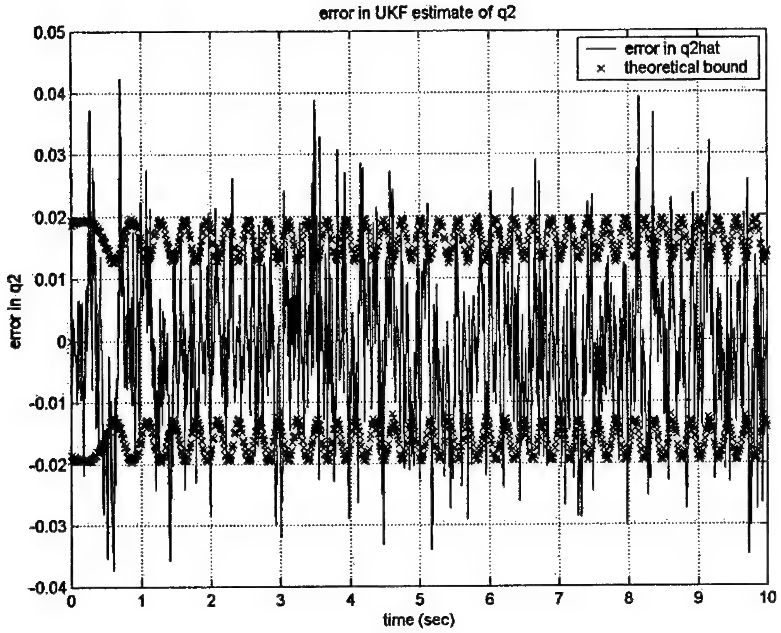


Figure C.11: Error in the UKF Estimate of the q_2 Component of the Quaternion

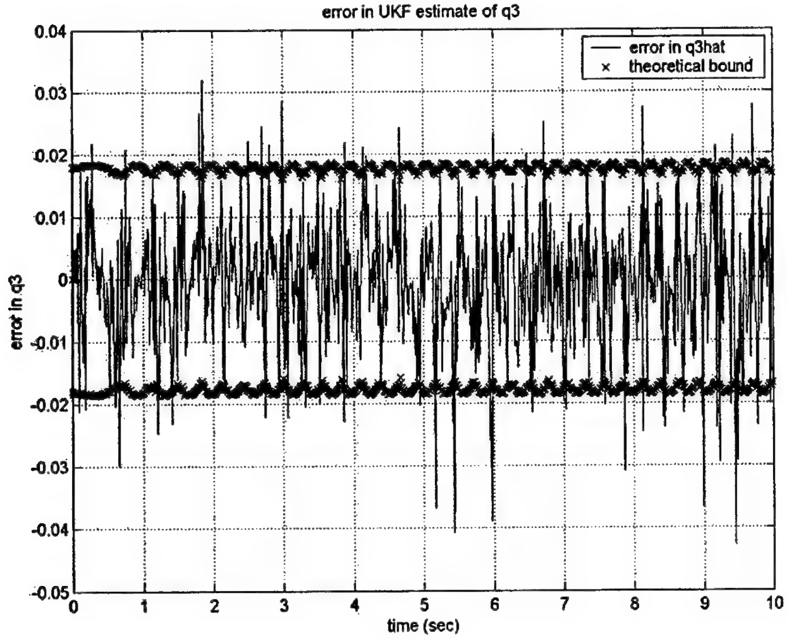


Figure C.12: Error in the UKF Estimate of the q_3 Component of the Quaternion

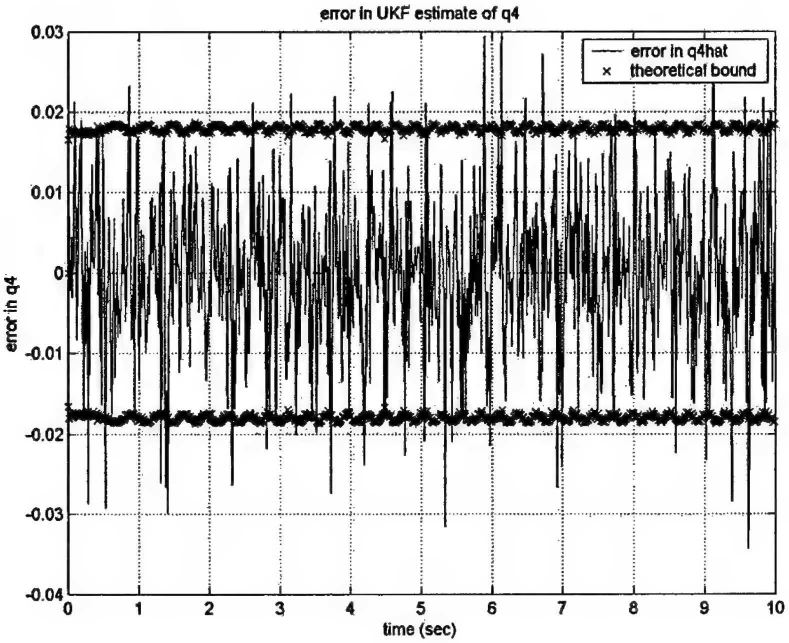


Figure C.13: Error in the UKF Estimate of the q_4 Component of the Quaternion

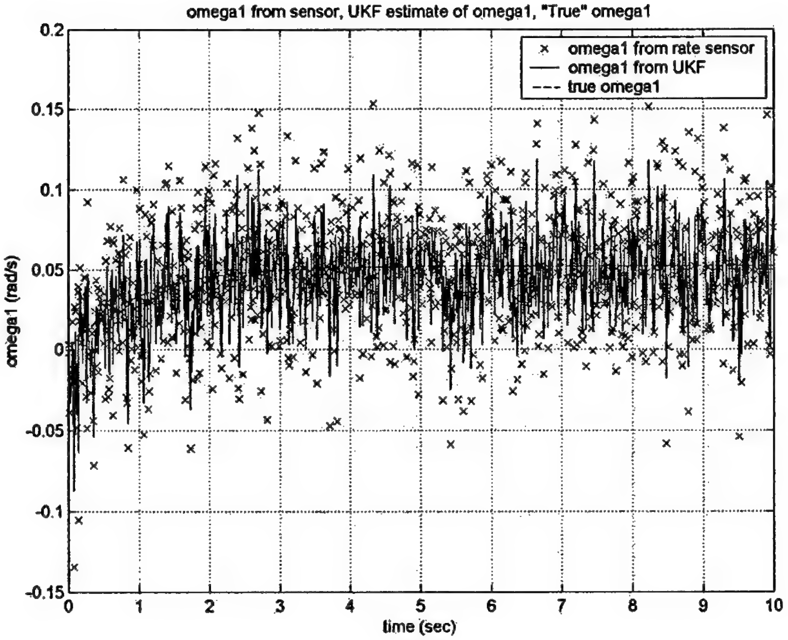


Figure C.14: UKF Estimate of ω_1 Versus Measured ω_1 Versus True ω_1

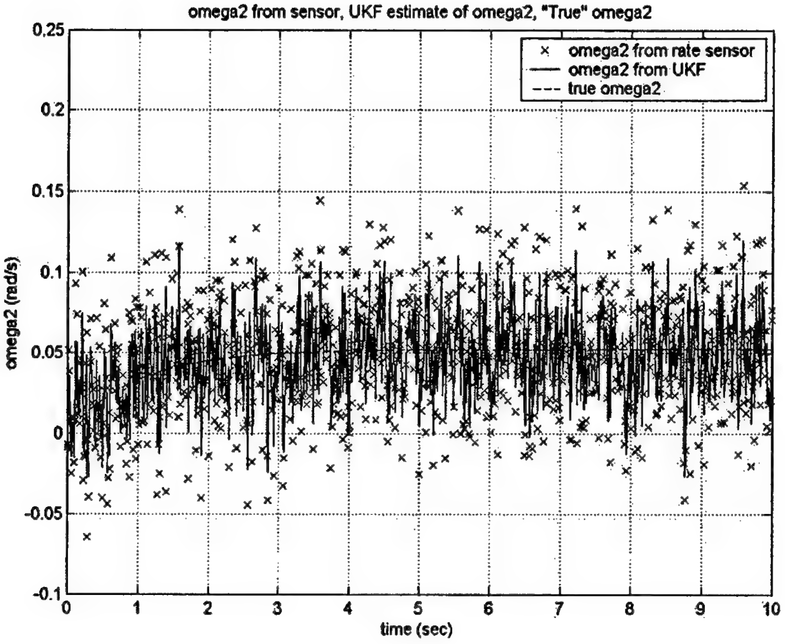


Figure C.15: UKF Estimate of ω_2 Versus Measured ω_2 Versus True ω_2

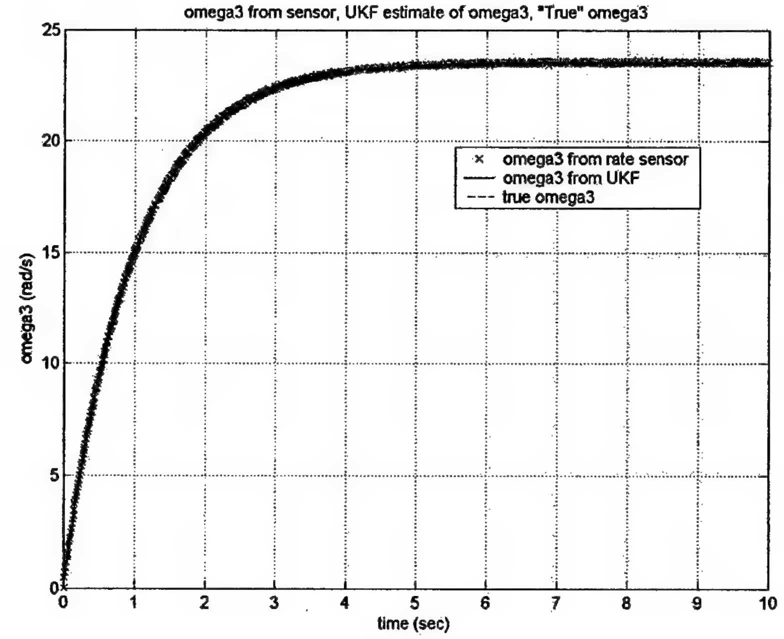


Figure C.16: UKF Estimate of ω_3 Versus Measured ω_3 Versus True ω_3

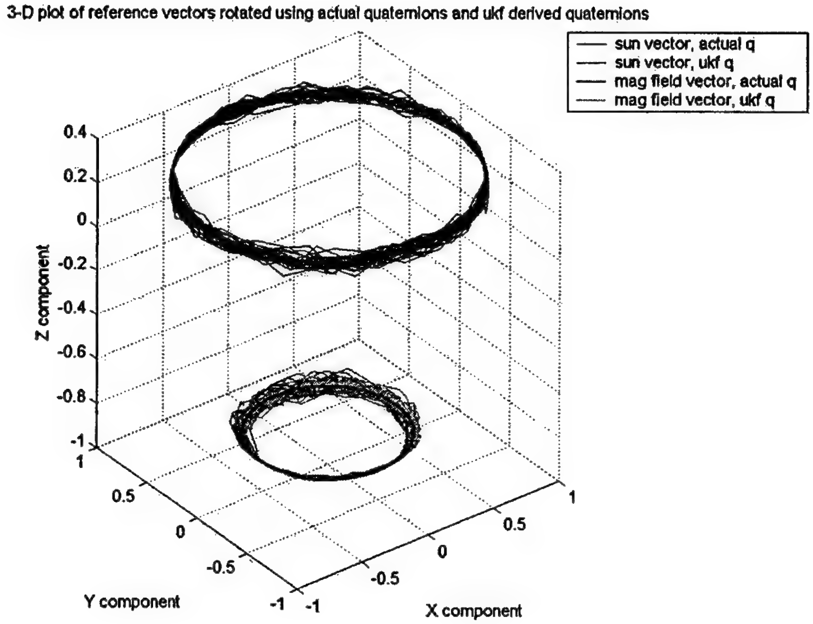


Figure C.17: Reference Vectors Rotated Using Gauss-Newton Derived and True Quaternions

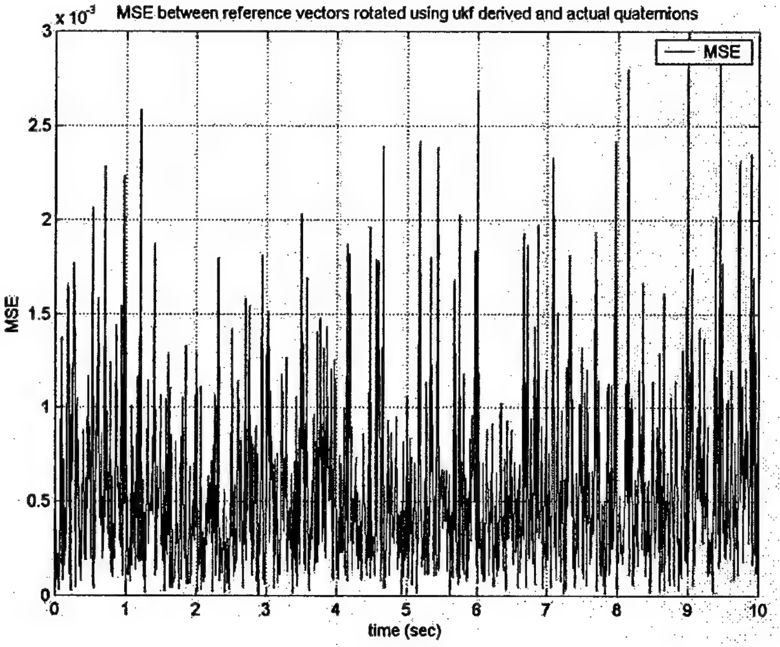


Figure C.18: MSE Between Vectors Rotated Using UKF derived and True Quaternions

UKF algorithm “with rates available.” Similar plots are generated for the EKF algorithms with and without rate measurements, and for the UKF algorithm without rates.

At the end of each filtering run, the user has the option to calculate the angles between the body axes rotated using the “true” quaternion and the body axes rotated using the filter estimated quaternion. In each case, the body frame is represented in inertial coordinates. A second option allows a choice of whether to plot the time step history of these results along with displaying the mean, standard deviation, and maximum of each, or to simply display the summary values. Figure C.19 is an example of the plot produced if that option is selected.

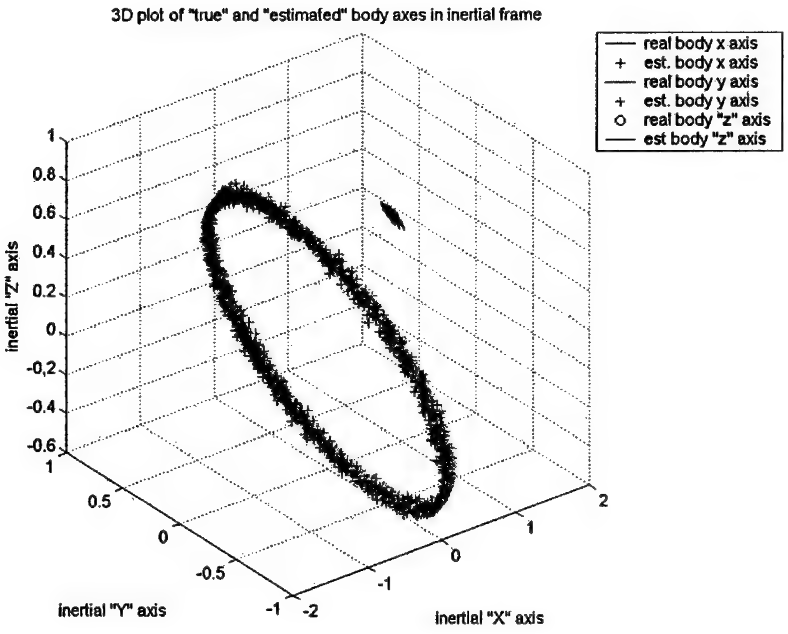


Figure C.19: 3D Plot of “True” and ”Estimated Body Axes in the Inertial Frame

Appendix D

Sample Results Used to Determine Process Noise and Time Constants

The following table contains the data generated while "tuning" the four filters as described in Chapter 10. This particular simulation was 40 seconds in duration and used the baseline sensor measurement standard deviations. The rate profile about each axis asymptotically approaches the baseline values of 0.5 rev/min about the x-axis, 0.5 rev/min about the y-axis, and 225 rev/min about the z-axis. The final rates are achieved at 20 seconds. All numerical integration was accomplished using a 0.01 sec step size unless otherwise noted in the table, and the UKF parameters α and β were set to 0.01 and 2 unless otherwise noted. GN-MSE is the average mean square error between the reference vector rotated using the true quaternion and that rotated using the Gauss-Newton generated quaternion. GN-FOM is the figure of merit for the Gauss-Newton routine as described in Chapter 10, PHIS is the process noise, Tau1/2 is the time constant corresponding to ω_1 and ω_2 while Tau3 is the time constant corresponding to ω_3. EKF-MSE and UKF-MSE are the average mean square error between the reference vector rotated using the filter and that rotated using the true quaternion for the EKF and the UKF respectively. EKF-FOM and UKF-FOM are the overall figure of merit, as defined in Chapter 10, for the EKF and the UKF respectively.

Table D.1: Sample Data From Tuning the Filters for One Simulation Case

Rate Profile: ***40 sec simulation, sun sensor sigma = 1.333 deg, mag sensor sigma = 3.333 deg, rate sensor sigma 0.333 rpm, asymptotic approach to final rpm at 20 sec, step size = .01 sec (unless otherwise noted by PHIS/timestep)***

Without rate measurements available

GN-MSE	GN-FOM	PHIS	Tau 1/2	Tau 3	EKF-MSE	UKF-MSE	EKF-FOM	UKF-FOM
0.0010	37.3090	1.86	31.00	1.00E+07	9.99E-04		0.1061	
0.0010	37.3090	3.80	28.00	1.00E+07	0.0010		0.0692	
0.0010	37.3090	3.82	28.00	1.00E+07	0.0010		0.0693	
0.0010	37.3090	3.78	28.00	1.00E+07	0.0010		0.0692	
0.0010	37.3090	3.75	28.00	1.00E+07	0.0010		0.0691	
0.0010	37.3090	3.60	28.00	1.00E+07	0.0010		0.0688	
0.0010	37.3090	3.55	28.00	1.00E+07	0.0010		0.0688	
0.0010	37.3090	3.50	28.00	1.00E+07	0.0010		0.0688	
0.0010	37.3090	3.40	28.00	1.00E+07	0.0010		0.0689	
0.0010	37.3090	3.41	28.00	1.00E+07	0.0010		0.0689	
0.0010	37.3090	3.45	28.00	1.00E+07	0.0010		0.0688	
0.0010	37.3090	3.44	28.00	1.00E+07	0.0010		0.0688	
0.0010	37.3090	3.43	28.00	1.00E+07	0.0010		0.0688	
0.0010	37.3090	3.42	28.00	1.00E+07	0.0010		0.0688	

Table D.1: Sample Data From Tuning the Filters for One Simulation Case (cont.)

0.0010	37.3090	3.65	28.00	1.00E+07	0.0010		0.0689	
0.0010	37.3090	3.61	28.00	1.00E+07	0.0010		0.0688	
0.0010	37.3090	3.62	28.00	1.00E+07	0.0010		0.0688	
0.0010	37.3090	3.63	28.00	1.00E+07	0.0010		0.0688	
0.0010	37.3090	3.64	28.00	1.00E+07	0.0010		0.0688	
0.0010	37.3090	3.53	31.00	1.00E+07	0.0010		0.0688	
0.0010	37.3090	3.53	34.00	1.00E+07	0.0010		0.0689	
0.0010	37.3090	3.53	33.00	1.00E+07	0.0010		0.0688	
0.0010	37.3090	3.53	25.00	1.00E+07	0.0010		0.0688	
0.0010	37.3090	3.53	20.00	1.00E+07	0.0010		0.0692	
0.0010	37.3090	3.53	24.00	1.00E+07	0.0010		0.0688	
0.0010	37.3090	3.53	23.00	1.00E+07	0.0010		0.0689	
0.0010	37.3090	3.53	29.00	1.00E+10	0.0010		0.0688	
0.0010	37.3090	3.53	29.00	1.00E+03	0.0010		0.0694	
0.0010	37.3090	3.53	29.00	5.00E+03	0.0010		0.0689	
0.0010	37.3090	3.53	29.00	1.00E+04	0.0010		0.0688	
0.0010	37.3090	3.53/.001	29.00	1.00E+07	0.0010		0.0688	
0.0010	37.3090	3.53/.0001	29.00	1.00E+07	0.0010		0.0688	
0.0010	37.3090	3.53/.1	29.00	1.00E+07	0.0010		3.963	
0.0010	37.3090	2.15	0.07	1.00E+07		0.0010		0.3151
0.0010	37.3090	2.00	0.07	1.00E+07		0.0010		0.2414
0.0010	37.3090	1.86	0.07	1.00E+07		0.0010		0.1849
0.0010	37.3090	1.00	0.07	1.00E+07		0.0010		0.0461
0.0010	37.3090	0.75	0.07	1.00E+07		0.0010		0.0887
0.0010	37.3090	0.90	0.07	1.00E+07		0.0010		0.054
0.0010	37.3090	1.10	0.07	1.00E+07		0.0010		0.0452
0.0010	37.3090	1.11	0.07	1.00E+07		0.0010		0.0454
0.0010	37.3090	1.09	0.07	1.00E+07		0.0010		0.045
0.0010	37.3090	1.08	0.07	1.00E+07		0.0010		0.0449
0.0010	37.3090	1.07	0.07	1.00E+07		0.0010		0.0449
0.0010	37.3090	1.06	0.07	1.00E+07		0.0010		0.0449
0.0010	37.3090	1.05	0.07	1.00E+07		0.0010		0.0449
0.0010	37.3090	1.04	0.07	1.00E+07		0.0010		0.045
0.0010	37.3090	1.1/.001	0.07	1.00E+07		0.0010		0.0452
0.0010	37.3090	1.1/.0001	0.07	1.00E+07		0.0010		0.0452
0.0010	37.3090	1.1/.1	0.07	1.00E+07		0.0010		0.0684
0.0010	37.3090	1.07	0.06	1.00E+07		0.0010		0.0449
0.0010	37.3090	1.07	0.05	1.00E+07		0.0010		0.0449
0.0010	37.3090	1.07	0.08	1.00E+07		0.0010		0.0449

Table D.1: Sample Data From Tuning the Filters for One Simulation Case (cont.)

0.0010	37.3090	1.07	0.04	1.00E+07		0.0010		0.00451
0.0010	37.3090	1.07	0.09	1.00E+07		0.0010		0.0449
0.0010	37.3090	1.07	0.1	1.00E+07		0.0010		0.0449
0.0010	37.3090	1.07	0.11	1.00E+07		0.0010		0.045
0.0010	37.3090	1.07	0.07	1.00E+05		0.0010		0.0449
0.0010	37.3090	1.07	0.07	1.00E+03		0.0010		0.0452
0.0010	37.3090	1.07	0.07	1.00E+10		0.0010		0.0449
0.0010	37.3090	1.07	0.07	1.00E+04		0.0010		0.0449
0.0010	37.3090	1.07	0.07	5.00E+03		0.0010		0.0449

With rate measurements available

GN-MSE	GN-FOM	PHIS	Tau 1/2	Tau 3	EKF-MSE	UKF-MSE	EKF-FOM	UKF-FOM
0.0010	0.0016	34.00	36.00	1.00E+07	0.001		0.0012	
0.0010	0.0016	36.00	36.00	1.00E+07	0.001		0.0011	
0.0010	0.0016	38.00	36.00	1.00E+07	0.001		0.0011	
0.0010	0.0016	40.00	36.00	1.00E+07	0.001		0.0011	
0.0010	0.0016	42.00	36.00	1.00E+07	0.001		0.0011	
0.0010	0.0016	44.00	36.00	1.00E+07	0.001		0.0011	
0.0010	0.0016	46.00	36.00	1.00E+07	0.001		0.0011	
0.0010	0.0016	48.00	36.00	1.00E+07	0.001		0.0011	
0.0010	0.0016	50.00	36.00	1.00E+07	0.001		0.0011	
0.0010	0.0016	52.00	36.00	1.00E+07	0.001		0.0011	
0.0010	0.0016	60.00	36.00	1.00E+07	0.001		0.0011	
0.0010	0.0016	65.00	36.00	1.00E+07	0.001		0.0011	
0.0010	0.0016	70.00	36.00	1.00E+07	9.80E-04		0.0011	
0.0010	0.0016	80.00	36.00	1.00E+07	9.78E-04		0.0011	
0.0010	0.0016	100.00	36.00	1.00E+07	9.73E-04		0.0011	
0.0010	0.0016	150.00	36.00	1.00E+07	9.69E-04		0.0011	
0.0010	0.0016	200.00	36.00	1.00E+07	9.68E-04		0.0012	
0.0010	0.0016	175.00	36.00	1.00E+07	9.68E-04		0.0011	
0.0010	0.0016	180.00	36.00	1.00E+07	9.68E-04		0.0011	
0.0010	0.0016	185.00	36.00	1.00E+07	9.68E-04		0.0011	
0.0010	0.0016	190.00	36.00	1.00E+07	9.68E-04		0.0012	
0.0010	0.0016	112.00	40.00	1.00E+07	9.72E-04		0.0011	
0.0010	0.0016	112.00	45.00	1.00E+07	9.72E-04		0.0011	
0.0010	0.0016	112.00	50.00	1.00E+07	9.72E-04		0.0011	
0.0010	0.0016	112.00	55.00	1.00E+07	9.72E-04		0.0011	
0.0010	0.0016	112.00	30.00	1.00E+07	9.72E-04		0.0011	
0.0010	0.0016	112.00	60.00	1.00E+07	9.72E-04		0.0011	
0.0010	0.0016	112.00	25.00	1.00E+07	9.72E-04		0.0011	
0.0010	0.0016	112.00	65.00	1.00E+07	9.72E-04		0.0011	
0.0010	0.0016	112.00	20.00	1.00E+07	9.72E-04		0.0011	

Table D.1: Sample Data From Tuning the Filters for One Simulation Case (cont.)

0.0010	0.0016	112.00	70.00	1.00E+07	9.72E-04		0.0011	
0.0010	0.0016	112.00	100.00	1.00E+07	9.72E-04		0.0011	
0.0010	0.0016	112.00	500.00	1.00E+07	9.72E-04		0.0011	
0.0010	0.0016	112.00	1.00E+07	1.00E+07	9.72E-04		0.0011	
0.0010	0.0016	38.00	1.00E+07	1.00E+07	6.05E-04		0.0012	
0.0010	0.0016	38.00	36.00	1.00E+04	0.001		0.0011	
0.0010	0.0016	38.00	36.00	1.00E+03	0.001		0.0011	
0.0010	0.0016	112.00	15.00	1.00E+07	9.72E-04		0.0011	
0.0010	0.0016	112.00	10.00	1.00E+07	9.72E-04		0.0012	
0.0010	0.0016	112.00	36.00	1.00E+10	9.72E-04		0.0011	
0.0010	0.0016	0.063	0.55	1.00E+07		8.60E-04		0.0012
0.0010	0.0016	0.100	0.55	1.00E+07		9.40E-04		0.0014
0.0010	0.0016	0.050	0.55	1.00E+07		7.93E-04		0.0011
0.0010	0.0016	0.040	0.55	1.00E+07		7.10E-04		9.38E-04
0.0010	0.0016	0.030	0.55	1.00E+07		5.78E-04		7.34E-04
0.0010	0.0016	0.020	0.55	1.00E+07		3.82E-04		6.95E-04
0.0010	0.0016	0.010	0.55	1.00E+07		3.55E-04		0.008
0.0010	0.0016	0.019	0.55	1.00E+07		3.59E-04		7.59E-04
0.0010	0.0016	0.021	0.55	1.00E+07		4.04E-04		6.59E-04
0.0010	0.0016	0.022	0.55	1.00E+07		4.36E-04		6.41E-04
0.0010	0.0016	0.023	0.55	1.00E+07		4.48E-04		6.36E-04
0.0010	0.0016	0.024	0.55	1.00E+07		4.68E-04		6.39E-04
0.0010	0.0016	0.023	0.60	1.00E+07		4.47E-04		6.36E-04
0.0010	0.0016	0.023	0.50	1.00E+07		4.48E-04		6.36E-04
0.0010	0.0016	0.023	0.40	1.00E+07		4.48E-04		6.36E-04
0.0010	0.0016	0.023	0.70	1.00E+07		4.48E-04		6.36E-04
0.0010	0.0016	0.023	0.30	1.00E+07		4.48E-04		6.36E-04
0.0010	0.0016	0.023	0.80	1.00E+07		4.48E-04		6.36E-04
0.0010	0.0016	0.023	5.00	1.00E+07		4.48E-04		6.38E-04
0.0010	0.0016	0.023	0.20	1.00E+07		4.48E-04		6.37E-04
0.0010	0.0016	0.023	0.10	1.00E+07		4.48E-04		6.48E-04
0.0010	0.0016	0.023	0.21	1.00E+07		4.48E-04		6.37E-04
0.0010	0.0016	0.023	0.19	1.00E+07		4.48E-04		6.37E-04
0.0010	0.0016	0.023	0.18	1.00E+07		4.48E-04		6.38E-04
0.0010	0.0016	0.010	0.18	1.00E+07		3.55E-04		0.0081
0.0010	0.0016	0.023	0.20	1.00E+10		4.48E-04		6.38E-04
0.0010	0.0016	0.023	0.20	1.00E+08		4.48E-04		6.37E-04
0.0010	0.0016	0.023	0.20	1.00E+09		4.48E-04		6.37E-04
0.0010	0.0016	0.023	0.20	1.00E+03		4.48E-04		6.37E-04
0.0010	0.0016	0.023	0.20	1.00E+02		4.48E-04		6.51E-04
0.0010	0.0016	0.023	0.20	5.00E+02		4.48E-04		6.39E-04
0.0010	0.0016	.023/.001	0.20	1.00E+07		4.48E-04		6.37E-04

Table D.1: Sample Data From Tuning the Filters for One Simulation Case (cont.)

0.0010	0.0016	.023/.0001	0.20	1.00E+07		4.48E-04		6.37E-04
0.0010	0.0016	.023/.1	0.20	1.00E+07		0.0831		0.0831

Same GN-EKF and GN-FOM as above, but varying α and β

α=.05	β=2	0.023	0.20	1.00E+07		4.48E-04		6.37E-04
α=.1	β=2	0.023	0.20	1.00E+07		4.48E-04		6.37E-04
α=.5	β=2	0.023	0.20	1.00E+07		4.48E-04		6.37E-04
α=1	β=2	0.023	0.20	1.00E+07		4.48E-04		6.37E-04
α=1e-4	β=2	0.023	0.20	1.00E+07		4.48E-04		6.37E-04
α=10	β=2	0.023	0.20	1.00E+07		4.48E-04		6.37E-04
α=.01	β=5	0.023	0.20	1.00E+07		4.48E-04		6.37E-04
α=.01	β=1	0.023	0.20	1.00E+07		4.48E-04		6.37E-04

Same data set as above, drop out all measurements for 1 sec beginning at 10 sec
Without rate measurements available

0.0114	39.5700	3.530	29.00	1.00E+07	0.0116		0.0707	
0.0114	39.5700	1.070	0.07	1.00E+07		0.0116		0.0570

Same data set as above, drop out all measurements for 1 sec beginning at 10 sec
With rate measurements available

0.0114	4.024	112.000	36.00	1.00E+07	0.0173		1.1627	
0.0114	4.024	0.023	0.20	1.00E+07		0.0103		0.4969

Same data set as above, bias of -0.1 rpm on all rate measurements
Without rate measurements available

0.0010	37.310	3.530	29.00	1.00E+07	0.0010		0.0688	
0.0010	37.310	1.070	0.07	1.00E+07		0.0010		0.0449

Same data set as above, bias of -0.1 rpm on all rate measurements
With rate measurements available

0.0114	0.0112	112.000	36.00	1.00E+07	9.70E-04		0.0108	
0.0114	0.0112	0.023	0.20	1.00E+07		0.0004		0.0105

Same data set as above, bias of +0.1 deg on each axis of sun sensor, and then vector normalized
Without rate measurements available

0.0035	40.440	3.530	29.00	1.00E+07	0.0035		0.2081	
0.0035	40.440	1.070	0.07	1.00E+07		0.0035		0.0952

Table D.1: Sample Data From Tuning the Filters for One Simulation Case (cont.)

Same data set as above, bias of +0.1 deg on each axis of sun sensor, and then vector normalized
With rate measurements available

0.0037
0.0037

0.0037	112.000	36.00	1.00E+07	0.0034		0.0035	
0.0037	0.023	0.20	1.00E+07		0.0029		0.0029